UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática

Universitat Politècnica de València

Desarrollo de aplicaciones para múltiples plataformas y uso de realidad virtual con Unity 3D

Proyecto Final de Carrera

Ingeniería informática

Autor: Alejandro Selma García

Director: Manuel Agustí Melchor

14 de junio de 2017

Índice general

Capítulo 1

Introducción y objetivos

1.1. Introducción

Unity es una de las herramientas de desarrollo de videojuegos y aplicaciones interactivas más utilizadas en la actualidad [1]. Uno de los principales motivos de su rápida aceptación es que ofrece soporte para una gran cantidad de plataformas, de manera que tras el desarrollo de cualquier aplicación es posible generar ejecutables para todas ellas sin tener que replantear el trabajo para cada una. Sin embargo, en el proceso de trasladar un videojuego o aplicación a otros dispositivos, es recomendable considerar sus características particulares para optimizar la experiencia a nivel de control o de rendimiento, entre otros.

Asimismo, Unity también da soporte a plataformas de realidad virtual, donde es especialmente interesante explorar la adaptación de aplicaciones existentes y probar con nuevas formas de control, según las posibilidades de cada dispositivo.

En este proyecto se discutirán las distintas plataformas soportadas por Unity, así como sus principales características como herramienta. Se abordará el desarrollo de una aplicación en 3D que pueda servir como base para un videojuego, y su distribución en varias plataformas, así como su posterior adaptación a dispositivos de realidad virtual.

El concepto de realidad virtual puede referirse a un conjunto de tecnologías bastante amplio, pero en este caso la adaptación estará especialmente enfocada a dispositivos móviles que se puedan emplear como visores que simulen un entorno tridimensional, con el que el usuario pueda interactuar de

diversas maneras. Una de las plataformas que hacen uso de esta tecnología es *Google Cardboard*, que permite emplear un teléfono móvil y un soporte de cartón, o visores de coste relativamente bajo, para acceder a la realidad virtual de una forma muy asequible. Esto contrasta con el coste de otros dispositivos mas avanzados, que utilizan computadores tradicionales o hardware específico para realizar el procesamiento necesario, y que tienen un coste habitualmente mas elevado.

Figura 1.1: Visores Cardboard. *Fuente: vr.google.com.*

1.2. Objetivos

Los principales objetivos de este proyecto son los siguientes:

- Introducir Unity como herramienta de desarrollo y sus plataformas soportadas, haciendo hincapié en el método de trabajo con esta herramienta y en sus ventajas, destacando su portabilidad.
- Mostrar el desarrollo paso a paso de una aplicación 3D que pueda servir como base para un videojuego, centrándose en la funcionalidad y considerando su rendimiento en dispositivos móviles. El objetivo es mostrar cómo se desarrollan los elementos principales de un videojuego con Unity, que consistirá en una escena en 3D en la que el jugador tendrá que derrotar a enemigos que recorren el escenario, evitando sus disparos. La finalidad no será obtener un juego con un acabado que permita su distribución inmediata en el mercado, sino centrarse en el propio desarrollo y en que pueda ser adaptado posteriormente a otras

plataformas, especialmente dispositivos móviles y visores de realidad virtual, y experimentar con sus posibilidades y métodos de control. Esto también condicionará el apartado gráfico, que no empleará modelos y texturas con un gran nivel de detalle, para evitar posibles problemas de rendimiento en dispositivos con menor capacidad de procesamiento u otras limitaciones. Otras posibles mejoras, como aumentar el número de niveles de juego o modos multijugador en red, quedarán fuera del ámbito del proyecto y se dejarán como futuras ampliaciones que se podrían abordar.

- Adaptar la aplicación desarrollada a dispositivos móviles que hagan uso de realidad virtual, especialmente a través de la plataforma *Cardboard* (figura 1.1). Se partirá de una versión de escritorio que pueda ser ejecutada en ordenadores personales, y a partir de ella se desarrollarán otras versiones para cubrir múltiples plataformas. Concretamente, además de la versión de escritorio, se implementará una versión para dispositivos móviles que haga uso de un acelerómetro, otra versión adaptada a la realidad virtual para visores de cartón Cardboard con un sólo gatillo, y otra para visores de realidad virtual que pueda manejarse con un controlador independiente.

- Mostrar el proceso de compilación y despliegue en varias plataformas distintas, así como evaluar la ejecución de la aplicación probándola en estas plataformas.

1.3. Alternativas

Se han considerado otros motores de desarrollo de videojuegos como alternativas a Unity, si embargo este último tiene algunas ventajas que lo hacen el más indicado para el ámbito de este proyecto.

Los motores considerados son *Unreal Engine 4*, desarrollado por Epic Games [2] (figura 1.2a) y *CryEngine*, desarrollado por Crytek [3] (figura 1.2b). Ambos tienen varios puntos en común, por ejemplo los dos son herramientas especialmente enfocadas al desarrollo de videojuegos con gráficos en 3D, y disponen de opciones más avanzadas en este aspecto para conseguir resultados más realistas y un mejor rendimiento en hardware de última generación.

Unity, por otra parte, está centrado en un método de desarrollo más ágil, basado en el uso del editor y scripts en C# que permiten hacer cambios y probarlos rápidamente, y por tanto tiene una curva de aprendizaje menos

(a) (b)

Figura 1.2: Logotipo de Unreal Engine 4 (a) y de CryEngine (b).

pronunciada.

En cuanto a las plataformas soportadas, actualmente los tres motores soportan tanto videoconsolas como PC, dispositivos móviles y realidad virtual, aunque en sus primeras versiones Unreal Engine y CryEngine estaban dirigidos principalmente a PC y videoconsolas.

El coste de las licencias de uso también se ha tenido en cuenta. CryEngine es actualmente una herramienta de código abierto y usarla es completamente gratuito. Unity y Unreal Engine también son gratuitos para un uso personal o académico, pero son de pago si se hace un uso profesional o se perciben ganancias por el software desarrollado. En el próximo capítulo se discutirán con más detalle las distintas licencias y plataformas disponibles en Unity.

1.4. Estructura de la memoria

A continuación se hará un resumen de los capítulos de los que consta esta memoria de proyecto, que cubrirán los objetivos planteados:

Descripción general de Unity En este capítulo se presenta el motor de videojuegos Unity3D, sus características principales y método de desarrollo.

Comparativa entre licencias y plataformas Se analizan las licencias de desarrollo disponibles para trabajar con Unity y todas las plataformas actualmente soportadas.

Desarrollo de una aplicación 3D con Unity Aquí se aborda el desarrollo de la versión de escritorio de la aplicación, que servirá como base

para el resto de versiones. Se muestra el desarrollo paso a paso, detallando las operaciones en el editor, manipulación de objetos y scripts empleados. Al final del capítulo se obtiene una versión con todos los elementos jugables que definen el juego, y una interfaz visual (HUD) con indicadores que muestran información acerca de la partida. Se usan formas geométricas básicas, sin materiales ni texturas.

Mejora del aspecto gráfico y efectos de sonido Se añaden texturas y materiales a ciertos elementos del juego, además de modelos 3D importados de herramientas externas (Blender) y de la Asset Store de Unity. También se incorporan sonidos, música, y un menú de pausa que permite ajustar algunas opciones.

Evolución de la realidad virtual y opciones disponibles En este capítulo se hace un repaso de la evolución de la realidad virtual a lo largo de la historia y se presentan las plataformas disponibles en la actualidad.

Adaptación de la aplicación a dispositivos de realidad virtual En este capítulo se parte de la aplicación original para adaptarla a dispositivos Cardboard de realidad virtual. Se desarrollan dos nuevas versiones: una para visores de cartón con un sólo gatillo, y otra pensada para ser utilizada con un controlador con varios botones. Se exploran los cambios necesarios en la propia aplicación para adaptarse a nuevos métodos de control en cada una de estas versiones, además de la utilización del SDK Google VR de Unity para lograr el efecto de inmersión de realidad virtual.

Generación de ejecutables para otras plataformas Se detalla el proceso de compilación y ejecución de la aplicación en varias plataformas, y se genera otra versión que utiliza como método de control el acelerómetro de un dispositivo Android o iOS.

Conclusiones y trabajos futuros En el capítulo final se hace un repaso general del trabajo realizado y de cómo se han cumplido los objetivos planteados, así como posibles ampliaciones que se podrían abordar para continuar con el proyecto en el futuro.

Capítulo 2

Descripción general de Unity

2.1. Presentación y características principales

Unity es un motor de videojuegos multiplataforma desarrollado por *Unity Technologies* (figura 2.1), que permite desarrollar para videoconsolas, PC, Mac, GNU/Linux, todo tipo de dispositivos móviles, la web, TV y dispositivos de realidad virtual (RV).

En un principio la propia herramienta fué anunciada únicamente para el sistema operativo OS X, sin embargo desde su lanzamiento en 2005 ha aumentado su soporte a más de 25 plataformas [4]. Por este motivo, además de otros que se mencionarán a continuación, Unity es una de las herramientas de desarrollo de videojuegos más utilizadas en la actualidad.

Figura 2.1: Logotipo de Unity.

Al ser un motor de videojuegos, Unity proporciona un conjunto de componentes y herramientas comunes listas para ser utilizadas, como por ejemplo *shaders*, simulación de físicas o efectos de iluminación, entre otros. Esto puede facilitar considerablemente la implementación respecto a un videojuego o aplicación desarrollada "desde cero", ya que no es necesario invertir tiempo en la preparación de estos componentes, que pueden ser necesarios o no según

el tipo de aplicación que se esté desarrollando.

Algunos ejemplos de tecnologías gráficas soportadas por el motor son: ajuste de resolución a la plataforma en la que se ejecuta la aplicación, compresión de texturas, soporte para mapeado topológico (*bump mapping*), mapeado por paralaje (*parallax mapping*), *reflection mapping*, efectos de oclusión ambiental o sombras dinámicas [5].

Unity permite desarrollar tanto en 3D como en 2D, además de proporcionar soporte para juegos y aplicaciones de realidad virtual y juego online multijugador.

2.2. Ventajas

Hay varias características que diferencian a Unity de otras alternativas, entre las que destacan especialmente un editor visual que facilita hacer cambios y probarlos de una forma muy productiva, y una gran portabilidad que permite generar ejecutables para casi cualquier plataforma, entre otras.

2.2.1. Entorno de trabajo

El método de trabajo con Unity está muy vinculado al editor visual, que es el elemento principal con el que interactúa el usuario. Este editor permite ver todos los elementos de la escena en detalle y hacer cambios con facilidad, a menudo sin que sea necesario escribir código. En otras herramientas similares el proceso de desarrollo puede ser más complicado, ya que habitualmente se requieren conocimientos de programación más específicos, además de que el manejo de la propia herramienta suele ser menos intuitivo.

La figura 2.2 muestra el entorno de trabajo por defecto, aunque todos los paneles que lo forman se pueden mover y redimensionar a gusto del usuario.

El panel *Scene* muestra la escena que está activa, y permite seleccionar los elementos que hay en ella con el botón izquierdo del ratón para ver sus propiedades o modificarlas. Existen varias opciones para mover la cámara por la escena, lo cual se puede hacer con los botones del ratón y con las teclas *Ctrl* y *Alt*. Con *Alt* y el botón izquierdo del ratón, por ejemplo, podemos rotar la cámara alrededor de un punto, y con la rueda podemos acercarla o alejarla. Junto a la pestaña *Scene* tenemos la pestaña *Game*, que nos permite ver el juego en ejecución. Esta pestaña se abre automáticamente al ejecutar

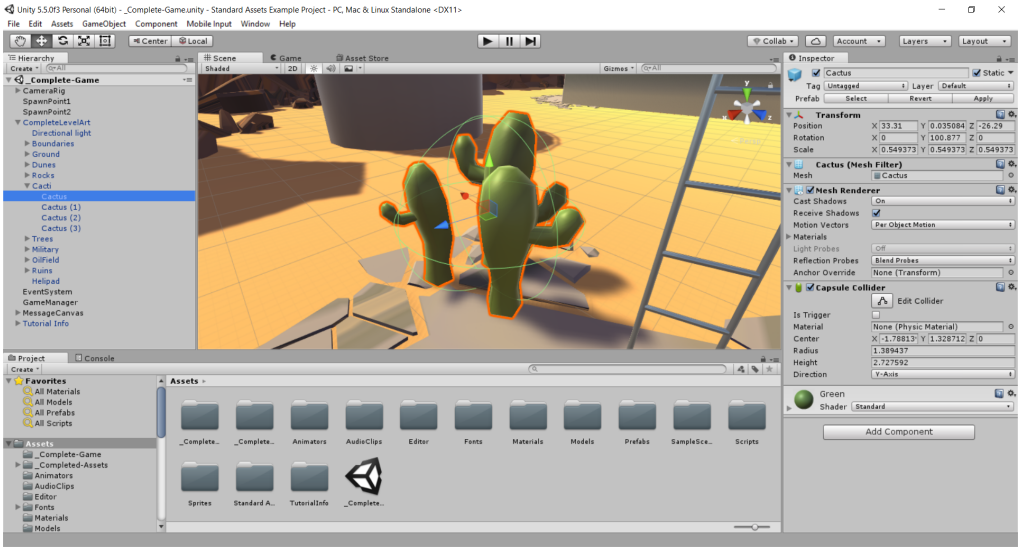


Figura 2.2: Entorno de trabajo de Unity.

la aplicación con el botón *Play*.

En el panel *Hierarchy* aparece una lista de todos los objetos en la escena, que en Unity se conocen como *GameObjects*. Como veremos más adelante, se pueden definir unos objetos como padres de otros, por tanto lo que se está representando es una jerarquía de objetos. Seleccionar un objeto en esta lista es equivalente a seleccionarlo directamente en la escena.

Al seleccionar un objeto podemos modificarlo de muchas maneras distintas. Entre las opciones más comunes están la traslación, la rotación y el escalado de objetos. Estas operaciones tienen sus botones correspondientes en la parte superior del área de trabajo, y pulsar uno u otro hará que se muestren en la escena indicadores que permiten mover el objeto, rotarlo o escalarlo.

Además, cuando tengamos un objeto seleccionado, aparecerá el panel *Inspector*. En este panel es donde podemos cambiar la mayoría de las propiedades de los objetos, que están agrupadas en *Components*. Cada uno de estos componentes define el comportamiento de cada objeto mediante propiedades relacionadas. Si queremos que un objeto se vea afectado por algunas leyes físicas, por ejemplo, se le puede añadir un componente de tipo *RigidBody* y cambiar estas propiedades en él. Los componentes se pueden añadir y eliminar libremente, a excepción de *Transform*, que está presente en todos los

objetos y contiene los vectores de posición, rotación y escalado mencionados anteriormente.

El inspector es uno de los elementos que facilita más el desarrollo, ya que permite cambiar rápidamente valores de variables y propiedades, incluso en tiempo de ejecución, sin tener que acceder al código. Sin embargo, hay ocasiones en las que si que hay que especificar parte de la lógica de la aplicación mediante código. Para ello se utilizan scripts, que representan comportamiento adicional definido mediante código en C# o *Javascript*. Una vez creado, un script es a su vez un *Component* que puede añadirse a uno o varios objetos de la escena.

Por estos motivos desarrollar en Unity es un proceso bastante fluido, ya que la mayor parte del tiempo se pueden mover y modificar los objetos directamente sobre la escena, cambiando su comportamiento a base de añadir los componentes adecuados y cambiar sus propiedades.

El último de los paneles principales es *Project*, donde se muestran todos los archivos del proyecto, incluyendo escenas, materiales, sonidos, imágenes, etc. En la pestaña Console podremos ver los mensajes de salida por consola, así como otros errores y avisos durante la ejecución.

2.2.2. Portabilidad

Otra de las razones de la rápida adopción de Unity como herramienta de desarrollo es su capacidad para generar ejecutables para múltiples plataformas, incluyendo las principales videoconsolas, ordenadores personales y dispositivos móviles. La lista ha ido aumentando con el tiempo y con la aparición de nuevo hardware. A continuación se muestra la lista de todas las plataformas soportadas oficialmente en la actualidad [4].

Ordenadores personales y web:

- Windows.
- Mac.
- Linux/Steam OS.
- WebGL.
- Facebook Gameroom.

Dispositivos móviles:

- iOS.
- Android.
- Windows Phone.
- Tizen.
- Fire OS.
- Windows Store Apps.

Realidad Virtual:

- Oculus Rift.
- Google Cardboard.
- Steam VR.
- Playstation VR.
- Gear VR.
- Microsoft Hololens.
- Daydream.

Videoconsolas:

- PlayStation 4.
- PlayStation Vita.
- Xbox One.
- WiiU.
- Nintendo 3DS.
- Nintendo Switch.

Smart TV:

- Android TV.
- Samsung SMART TV.
- tvOS.

2.2.3. Documentación

Hay mucha documentación disponible acerca de Unity en su página web, así como todo tipo de tutoriales y guías para tareas específicas [6]. Estos contenidos van dirigidos a usuarios de todos los niveles. Otra ventaja importante es que la propia comunidad de usuarios es muy amplia, por lo que se pueden resolver dudas específicas tanto en los foros de la web oficial como en otros sitios de internet.

2.2.4. Asset Store e importación de recursos

La Asset Store es un repositorio de contenidos creados por la comunidad que pueden importarse directamente en un proyecto de Unity, como modelos, imágenes o sonidos. Algunos de estos recursos son gratuitos y otros son de pago, y existen varias licencias de uso distintas según el contenido. La utilización de estos recursos puede facilitar el desarrollo considerablemente.

Capítulo 3

Comparativa entre licencias y plataformas soportadas

A continuación se compararán las distintas licencias de uso existentes en Unity y se describirán brevemente todas las plataformas para las que se puede desarrollar con esta herramienta.

3.1. Licencias de uso

Unity se puede utilizar en la actualidad con cuatro licencias distintas: Personal, Plus, Pro y Enterprise.

Personal Esta es la licencia más básica, e incluye todas las prestaciones del motor y soporte para todas las plataformas. Usar esta licencia no tiene ningún coste, por lo que es un buen punto de partida para usuarios que estén empezando o que no tengan previsto hacer un uso profesional de la herramienta. Las principales diferencias respecto a licencias superiores son otros servicios ofrecidos (como Unity Analitics) y un límite a los ingresos anuales de $100.000.

Plus La licencia Plus incluye todo lo ofrecido en la licencia Personal, además de la posibilidad de cambiar o eliminar la pantalla con el logotipo de Unity que aparece por defecto al lanzar la aplicación. También ofrece una versión más completa de Unity Analitics, capacidad para más usuarios simultáneos alojados por Unity en modos multijugador online, otros servicios como informes de ejecución, y descuentos en algunos paquetes de assets. El límite de ingresos anuales aumenta a $200.000. El coste de esta licencia es de $35 por puesto/mes.

Pro Esta licencia está dirigida a un uso profesional. Incluye todo lo ofrecido en la licencia Plus mejorando aún más los servicios ofrecidos. Elimina el límite a los ingresos anuales y permite llegar a un acuerdo con Unity Technologies para acceder al código fuente del motor y contratar soporte Premium, que es un servicio de asesoramiento. El coste de esta licencia es de $125 por puesto/mes.

Enterprise Es una licencia para organizaciones, que permite obtener servicios personalizados. No se indica el coste de esta licencia, ya que la compañía insta a ponerse en contacto con ella para llegar a un acuerdo respecto al coste y los servicios contratados.

La figura 3.1 muestra un resumen de todo lo ofrecido en cada licencia.

Compara los planes	Personal	Plus	Pro	Enterprise
Accelerator Pack		Gratis (con un valor de $190)	Gratis (con un valor de $190)	
Todas las prestaciones del motor	✓	✓	✓	✓
Todas las plataformas	✓	✓	✓	✓
Actualizaciones continuas	✓	✓	✓	✓
Sin regalías	✓	✓	✓	✓
Pantalla de inicio	Pantalla de inicio MWU	Animación personalizada o ninguna	Animación personalizada o ninguna	Animación personalizada o ninguna
Capacidad de ingresos	$100 mil	$200 mil	Ilimitado	Ilimitado
Cloud Build de Unity	Cola estándar	Cola de prioridades	Compilaciones simultáneas	Agentes de compilaciones dedicados
Unity Analytics	Analytics Personal	Plus Analytics	Analytics Pro	Analytics personalizado
Unity Multiplayer	20 usuarios simultáneos	50 usuarios simultáneos	200 usuarios simultáneos	Multiplayer personalizado
Compras dentro de la aplicación de Unity	✓	✓	✓	✓
Unity Ads	✓	✓	✓	✓
Acceso a Beta	✓	✓	✓	✓
Editor UI Skin de la versión Pro		✓	✓	✓
Informes de ejecución		✓	✓	✓
Gestión de puestos flexible		✓	✓	✓
Kits de assets		20 % de descuento	40 % de descuento	40 % de descuento
Software educativo para certificación de Unity		Acceso de 1 mes	Acceso de 3 meses	Acceso por 3 meses
Acceso al código fuente			$	$
Soporte Premium			$	$

Figura 3.1: Licencias disponibles en Unity. *Fuente: store.unity.com.*

3.2. Plataformas

3.2.1. Windows, MacOS y GNU/Linux

Unity permite generar ejecutables para los tres principales sistemas operativos presentes en ordenadores personales. En el caso de Windows, el motor puede utilizar las APIs gráficas Direct3D, OpenGL y Vulkan a partir de la versión 5.6 [7]. Para MacOS sólo puede utilizarse OpenGL, mientras que en GNU/Linux se pueden usar tanto OpenGL como Vulkan.

3.2.2. WebGL

WebGL es una API implementada en Javascript que, mediante el uso de HTML 5, permite la ejecución de aplicaciones 3D en un navegador web (figura 3.2). Unity permite generar ejecutables para esta plataforma, a través de compiladores que transforman el código nativo en .NET, C# y UnityScript en ficheros en C++, que finalmente se transforman en el código Javascript necesario [8].

Figura 3.2: Logotipo de WebGL.

3.2.3. iOS

Para el desarrollo de juegos o aplicaciones en dispositivos móviles, hay que tener en cuenta que el hardware está más estandarizado que en el caso de los ordenadores personales, además de que suele tener menor potencia para procesar gráficos porque se usan chips integrados en lugar de tarjetas gráficas dedicadas. Por ello hay que abordar el desarrollo de aplicaciones para sistemas operativos como iOS o Android de forma algo diferente. Esto incluye, por ejemplo, acceder a funcionalidades como la pantalla táctil o el GPS, y tener precaución con el consumo de energía derivado del uso de la aplicación.

En el caso concreto de Unity para iOS hay otras diferencias. Si se utiliza código en Javascript, el tipado dinámico estará desactivado por defecto [9], ya que si sólo se utiliza tipado estático el rendimiento mejora considerablemente

en dispositivos con iOS. Esto puede causar errores al importar proyectos con código existente en Javascript, y puede solucionarse indicando explícitamente el tipo de las variables. Por otra parte, al importar ficheros de audio, vídeo o texturas, se aplica una conversión a tipos de archivo soportados por la plataforma, si es posible. Algunos formatos, como texturas en DXT, no son soportadas en iOS.

3.2.4. Android

Android (figura 3.3) es un sistema operativo de código abierto basado en el kernel de Linux y desarrollado por Google. Es uno de los sistemas más utilizados en dispostivos móviles, y Unity permite desarrollar aplicaciones nativas para él, además de aplicaciones que hagan uso de realidad virtual mediante plataformas como *Cardboard* (también soportada en iOS) o *Daydream*. La API utilizada es OpenGL.

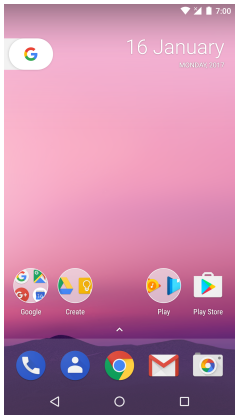


Figura 3.3: Android 7.1 Nougat. *Fuente: wikipedia.org.*

Las consideraciones a tener en cuenta son similares a las de desarrollar para iOS, si bien el hardware está menos estandarizado. Compilar para Android también requiere tener instalados el *Java Development Kit* (JDK), el propio SDK de Android y en caso de que se utilice IL2CPP, el *Android Native Development Kit* (NDK) [10].

3.2.5. Windows Store Apps para Windows Phone y HoloLens

Con Unity también se pueden desarrollar aplicaciones para la Windows Store (figura 3.4) y ejecutarlas en Windows, en dispositivos móviles con Win-

dows Phone o en visores *HoloLens*. El proceso de compilación y distribución implica desarrollar la aplicación en Unity, y seleccionar como plataforma de destino *Windows Store Apps*. Esto genera un proyecto de Visual Studio, con el que se puede compilar y generar el ejecutable final.

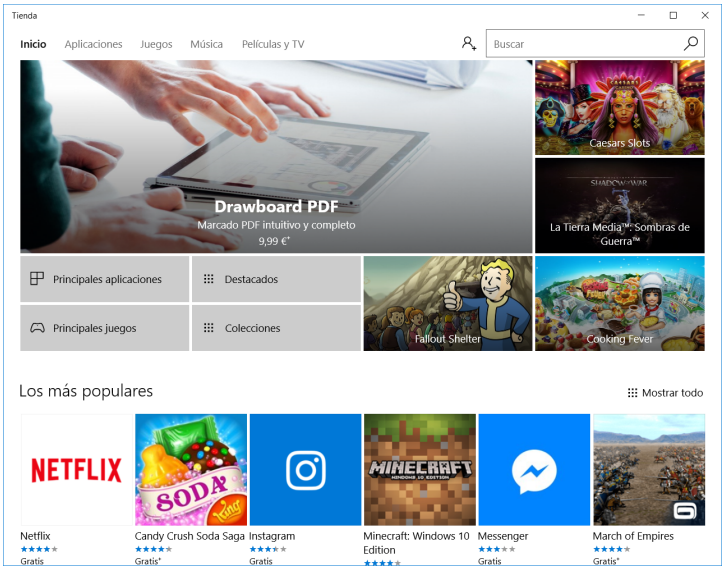


Figura 3.4: Windows Store.

Los requisitos necesarios para desarrollar en estas plataformas dependen del SDK de Windows con el que funcionará la aplicación:

Para el *SDK Windows 8.1*, *Windows Phone 8.1* o *Universal SDK 8.1*, es necesario realizar la compilación en Windows 8.1 o superior y disponer de Visual Studio 2013 o alguna versión posterior.

Para el *SDK Universal Windows 10 Apps*, se requiere Windows 8.1 o superior, Visual Studio 2015 o posterior y tener instalado el propio *SDK Universal Windows 10 Apps*.

Las aplicaciones para *HoloLens* son consideradas como *Universal Windows Applications* por Microsoft, de manera que los procedimientos de desarrollo y requisitos son los mismos que los de cualquier aplicación de Windows Store.

3.2.6. Dispositivos de realidad virtual

Unity permite desarrollar aplicaciones para los siguientes dispositivos de realidad virtual: Oculus Rift, Steam VR, Google Cardboard, Daydream, PlayStation VR, Gear VR y Microsoft Hololens (figura 3.5). Estas y otras plataformas de realidad virtual se tratarán en detalle en su propio capítulo.

Figura 3.5: Plataformas de realidad virtual soportadas por Unity. *Fuente: unity3d.com.*

3.2.7. Tizen

Tizen (figura 3.6) es un sistema operativo libre basado en el kernel de Linux, y construido inicialmente a partir de la plataforma Linux de Samsung (SLP). Funciona en todo tipo de dispositivos, incluyendo PC, dispositivos móviles o Smart TVs.

Figura 3.6: Tizen 2.2.

Unity permite generar aplicaciones ejecutables para Tizen en smartphones Samsung. El único requisito es instalar el SDK para el desarrollo de aplicaciones en esta plataforma, y otro SDK para la gestión de certificados (*Tizen*

(a) (b)

Figura 3.7: Videoconsolas PlayStation 4 (a) y Nintendo 3DS (b).

Certificate Extension SDK), que son necesarios para instalar aplicaciones en el sistema.

3.2.8. Fire OS

Fire OS es un sistema operativo para móviles desarrollado por Amazon y basado en Android. Sus principales diferencias con Android son una interfaz de usuario personalizada y el uso de aplicaciones propias de Amazon que sustituyen al software de Google y que promocionan el uso de sus propios servicios. Las aplicaciones ejecutadas son de tipo *Android Package* (APK), por tanto distribuir aplicaciones para esta plataforma a través de Unity sigue el mismo procedimiento que hacerlo para sistemas Android.

3.2.9. Videoconsolas

Actualmente se puede desarrollar con Unity para PlayStation 4 (figura 3.7a), Playstation Vita, Xbox 360, Xbox One, Wii, WiiU, Nintendo 3DS (figura 3.7b) y Nintendo Switch. Por lo general, desarrollar aplicaciones para estas plataformas requiere publicarlas pasando por un proceso de aprobación que varía según el titular de cada una. En el caso de WiiU, Unity es la herramienta por defecto que viene incluida con cada licencia de desarrollo para esta consola [11].

3.2.10. Android TV

Android TV es un sistema operativo basado en el kernel de Linux, y orientado a ser utilizado en televisores. Este sistema permite utilizar juegos y aplicaciones en televisores con una interfaz similar a la de Android en teléfonos móviles. El procedimiento para generar ejecutables en esta plataforma es el mismo que para Android, ya que se trata prácticamente del mismo sistema operativo y las aplicaciones se instalan a partir de archivos en formato APK. En las opciones de compilación de Unity hay que asegurarse que estén marcadas las opciones *Android TV Compatibility* y *Android Game* (figura 3.8), en caso de que se trate de un juego, para que se pueda utilizar el mando del televisor u otro dispositivo como controlador.

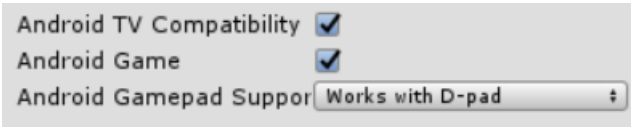


Figura 3.8: Opciones de compilación para Android TV.

3.2.11. tvOS

tvOS es un sistema operativo desarrollado por Apple pensado para funcionar en cualquier televisor de alta definición a través del reproductor Apple TV (figura 3.9), que incluye un controlador. Esta basado en iOS, y permite ver contenidos de servicios de streaming de vídeo y ejecutar aplicaciones descargadas de la Apple Store. Cualquier aplicación para iOS puede ejecutarse en tvOS, si bien puede ser necesario hacer algunos cambios en la aplicación, como adaptarla a una pantalla más grande, para que la experiencia sea óptima. En el caso de videojuegos, suele ser necesario adaptar los controles al controlador de Apple TV.

Unity permite generar ejecutables para tvOS seleccionándola como plataforma a la hora de compilar. Sin embargo hay algunos requisitos: es necesario disponer del dispositivo Apple TV de 4ª generación, el cable de conexión y el IDE *XCode* en su versión 7.1 o superior [12].

3.2.12. Samsung SMART TV

Unity también ofrece la posibildad de desarrollar aplicaciones para televisores Samsung SMART TV. Estos televisores tienen como hardware un

Figura 3.9: Dispositivo Apple TV. *Fuente: apple.com.*

procesador ARM y GPU con OpenGLES, por lo que presentan un rendimiento similar al de los dispositivos móviles. Para realizar la compilación desde Unity es necesario introducir la dirección IP del televisor, que puede obtenerse fácilmente a través de la interfaz de este.

Capítulo 4

Desarrollo de una aplicación 3D con Unity

A continuación se mostrará paso a paso cómo desarrollar una aplicación que pueda servir como base para un videojuego en 3D, con vistas a que pueda ser adaptada más adelante a un entorno de realidad virtual. El objetivo es construir una escena a partir de formas básicas, en la que el jugador se pueda mover libremente e interactuar con elementos del escenario.

4.1. Creación de un nuevo proyecto

Para empezar hay que iniciar Unity y seleccionar *New* (figura 4.1) para introducir el nombre del proyecto y la ruta donde se guardarán los ficheros necesarios. En este caso marcaremos la casilla 3D, ya que la aplicación va a ser de este tipo. Esto influye en la configuración inicial del proyecto, que quedará preparada para trabajar en un entorno tridimensional.

Tras pulsar *Create Project*, aparecerá la pantalla principal de Unity con una escena vacía, a excepción de la cámara principal y una fuente de luz direccional (figura 4.2). Si seleccionamos la cámara que se ha creado por defecto, podemos verificar en el inspector que tiene el atributo *Projection* establecido en modo perspectiva. Este es el modo que debe usarse en escenas en 3D, ya que permite apreciar la profundidad de los objetos (objetos más lejanos respecto a la cámara se verán más pequeños, mientras que los más cercanos se verán más grandes). El otro modo que hay disponible, que establece una cámara ortográfica, elimina la sensación de profundidad y se usa habitualmente para aplicaciones en 2D.

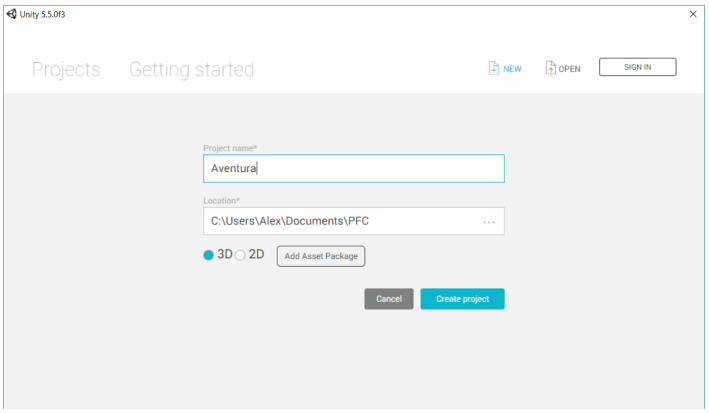


Figura 4.1: Pantalla de Nuevo proyecto.

En cuanto a la fuente de luz, se puede ver en el inspector que es de tipo direccional. Este tipo de fuentes de luz proyectan los rayos de forma paralela, por lo que se usan para emular la iluminación global de la escena.

4.2. Objetos y sistemas de coordenadas

Es importante mencionar que prácticamente cualquier objeto que se incluya en la escena en Unity es de tipo *GameObject*. Estos objetos tienen asociados una serie de componentes (*Components*), que definen las propiedades del objeto. Todos los objetos tienen el componente *Transform*, que define la posición del objeto en la escena, su rotación y su escalado. Estos tres atributos vienen indicados como vectores de 3 coordenadas.

El sistema de coordenadas es muy similar al de otras herramientas de desarrollo o modelado en 3D. Respecto al origen, el eje X es el eje horizontal, de forma que incrementar o decrementar este valor en el vector de posición desplaza el objeto hacia la derecha o a la izquierda respectivamente. El eje Y es el eje vertical e indica la altura, valores más altos la aumentan y valores más bajos la disminuyen. Por último, el eje Z indica la profundidad. Aumentar este valor mueve el objeto hacia adelante, y disminuirlo lo mueve hacia atrás.

A este sistema también se le conoce como sistema de coordenadas levógiro, ya que las direcciones positivas (crecientes) de los 3 ejes coinciden con los dedos pulgar, índice y anular de la mano izquierda, si se colocan en ángulo

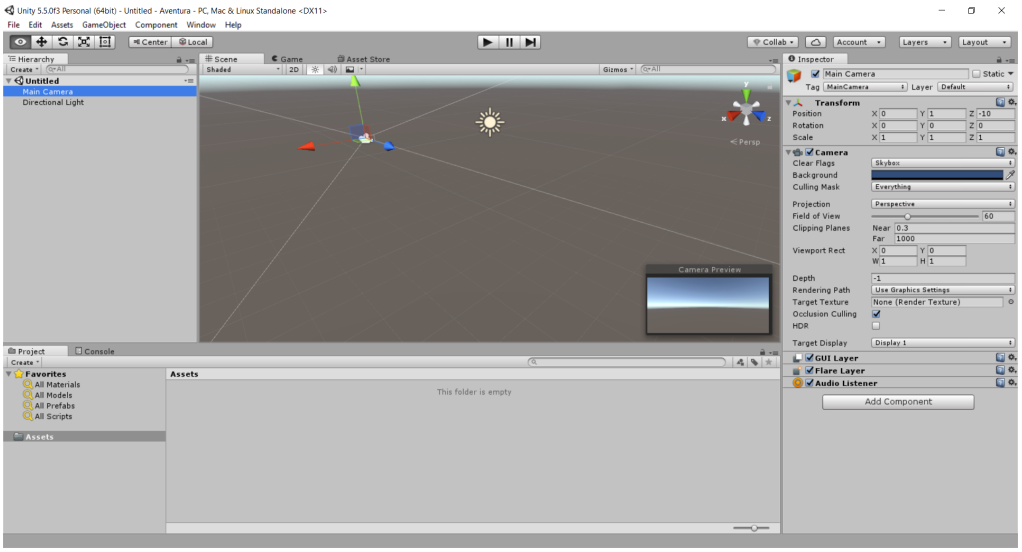


Figura 4.2: Pantalla principal de Unity.

recto. Otras herramientas utilizan un sistema de coordenadas dextrógiro, en el que el eje Z tiene sentido contrario (figura 4.3) [14].

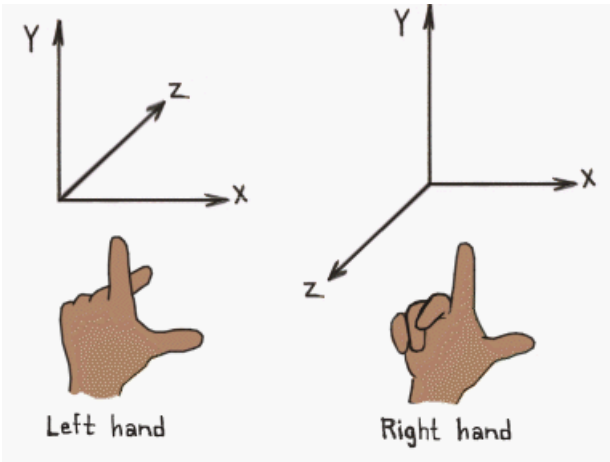


Figura 4.3: Sistemas de coordenadas levógiro y dextrógiro.

En cuanto a los vectores de rotación y escalado, estos indican la rotación en grados y el escalado que se aplicará respecto a cada uno de los 3 ejes.

4.3. Construcción del escenario

Para construir la escena, empezaremos añadiendo cubos y aplicando transformaciones para formar el suelo y las paredes. Para ello, seleccionamos *GameObject* → *3D Object* → *Cube*, y aparecerá un cubo en el centro de la escena (figura 4.4).

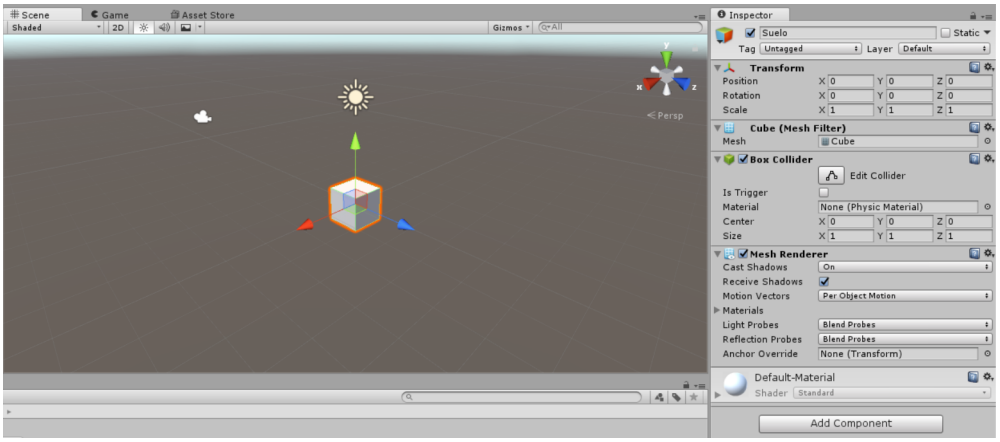


Figura 4.4: GameObject de tipo Cube recién creado.

En el inspector se puede cambiar el nombre del objeto (lo renombraremos como *Suelo*) y se pueden ver y manipular los componentes del mismo. En este momento nos interesa redimensionar el cubo para formar el suelo, así que cambiaremos los valores del vector de escalado para aumentar su anchura y profundidad. Para ello, podemos seleccionar la herramienta de escalado que se encuentra en la parte superior izquierda y a continuación redimensionarlo arrastrando los 3 ejes situados sobre él, o bien modificar los valores directamente en el vector *Scale* del componente *Transform*. Inicialmente le daremos el valor 100 a las componentes X y Z del vector, dejando la altura al valor 1 que viene por defecto, si bien podremos cambiar estos valores más adelante según sea necesario (figura 4.5).

A continuación se seguirá el mismo procedimiento para situar el resto de los elementos más básicos de un escenario que el jugador pueda recorrer: unas paredes exteriores que sirvan de límite, muros para separar espacios o formar pasillos y algún elemento decorativo como una casa o una torre.

Esta técnica, conocida como *whiteboxing* o *greyboxing* [15], nos permite construir rápidamente un escenario a base de formas simples. Se utiliza

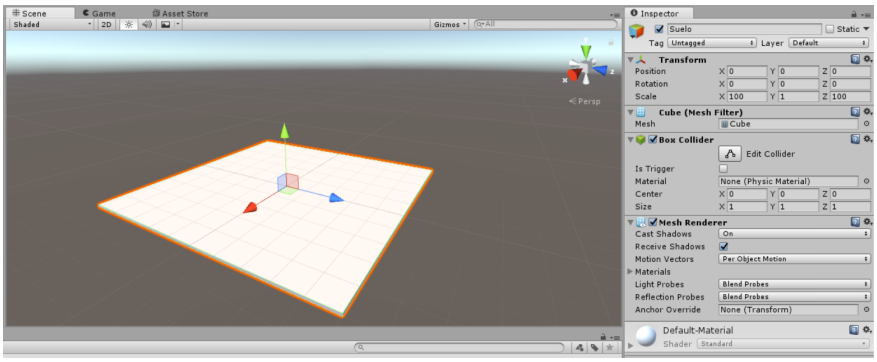


Figura 4.5: El cubo redimensionado forma el suelo del escenario.

bastante en el desarrollo de videojuegos para hacer prototipos de niveles y poder hacer todo tipo de pruebas sin tener que preocuparse del acabado final.

Para colocar una de las paredes externas, por ejemplo, se puede añadir otro cubo y situarlo 1 unidad por encima del suelo, darle un escalado en el eje Y de 2 y desplazarlo en el eje Z la longitud de uno de los lados del escenario dividida entre 2 (si se encuentra situado inicialmente en el origen), tal y como muestra la figura 4.6. Para las demás paredes el proceso sería parecido, y ya que las dimensiones no van a cambiar, se puede duplicar el objeto recién creado seleccionándolo y pulsando Ctrl+D. Después basta con cambiar el valor de la profundidad en el nuevo objeto para colocar una pared idéntica en el lado opuesto. Las dos paredes restantes habrá que rotarlas 90º respecto al eje Y, y desplazarlas en el eje X para colocarlas en su sitio.

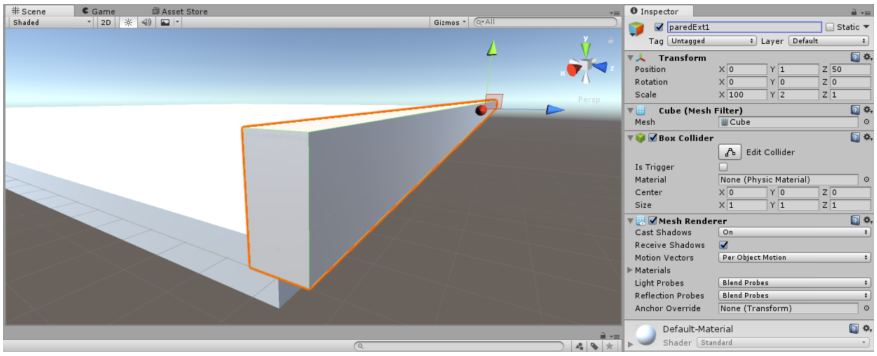


Figura 4.6: Pared externa.

Además de las paredes, añadiremos muros en el interior y otros elementos, usando trasformaciones, rotaciones y escalados como se ha hecho anteriormente. El resultado es el que se muestra en la figura 4.7.

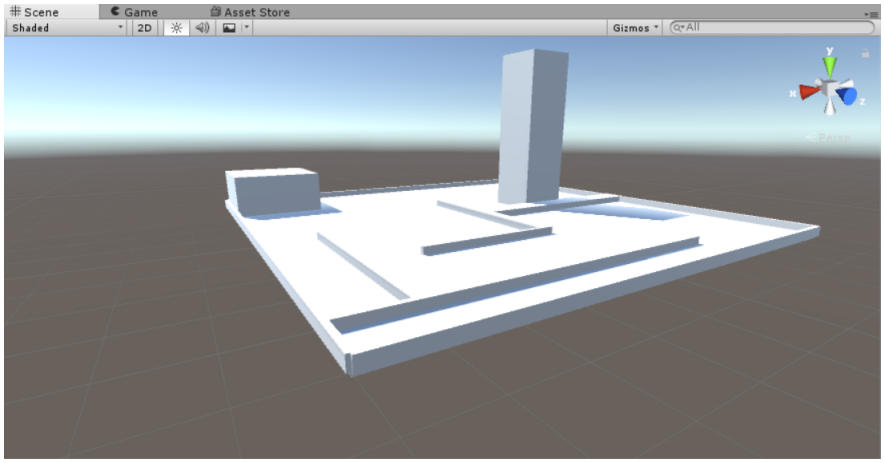


Figura 4.7: Escenario compuesto de formas básicas.

4.4. Iluminación de la escena

En estos momentos la iluminación viene de la única fuente de luz que venía incluida al crear el proyecto. Puesto que la luz es de tipo direccional, se considera que está a una distancia infinita y su posición no influye en la iluminación de la escena. Sin embargo, se puede ajustar su rotación para establecer la inclinación de los rayos, así como la intensidad de la fuente de luz. Ajustaremos la intensidad a 0.5 y también cambaremos la inclinación para poder apreciar los cambios en el escenario.

Puesto la intensidad de la iluminación global se ha reducido, añadiremos algunas fuentes de luz posicionales del tipo *Point Light* (*GameObject* → *Light* → *Point Light*). En este tipo de fuente de luz la posición sí es importante, ya que la iluminación disminuye con la distancia. Este valor se puede ajustar con el atributo *Range*, así como la intensidad. Las posicionaremos en algunos de los elementos del escenario para resaltarlos más, o en zonas que no estén muy bien iluminadas. A pesar de todo, es importante tener en cuenta que muchas fuentes de luz pueden afectar al rendimiento de la aplicación. En la 4.8 se puede apreciar los cambios y las propiedades de una de las luces posicionales.

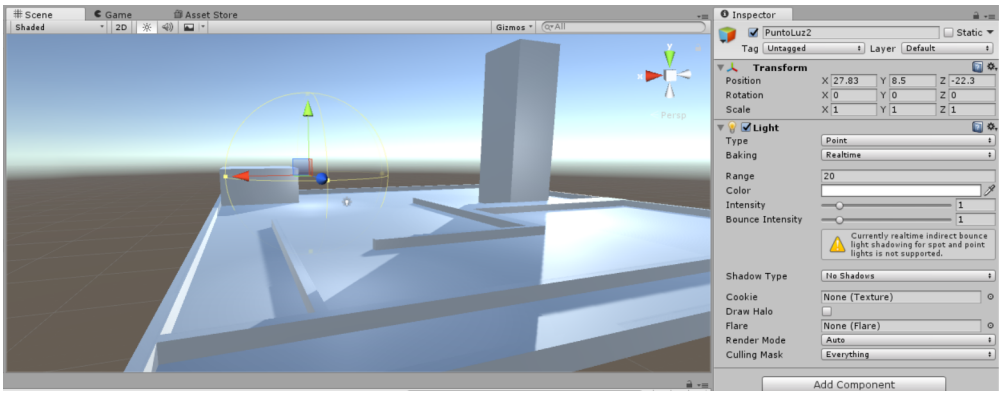


Figura 4.8: Escenario con fuentes de luz posicionales.

4.4.1. Organizando la jerarquía de objetos

En estos momentos, todos los elementos que se han ido añadiendo a la escena aparecen en la sección *Hierarchy*. A medida que se vayan añadiendo más, puede resultar confuso tener tantos objetos listados, por eso resulta conveniente agrupar objetos relacionados. Para ello podemos crear un nuevo objeto vacío, que llamaremos Escenario (*GameObject* → *Create Empty*), y arrastrar dentro de él todos los objetos que formen parte del escenario. De esta forma quedarán agrupados como objetos "hijos"de *Escenario*, que será el objeto padre (4.9).

Esto no sólo es útil para mantener organizada la lista de objetos, también afecta a sus propiedades. Cuando los objetos forman una jerarquía, los hijos heredan algunas propiedades del objeto padre, como por ejemplo el componente *Transform*. Esto significa que su posición, rotación y escalado ya no están definidos en relación al origen, sino en relación al objeto padre. Cualquier cambio de este tipo que afecte al padre afectará a todos sus hijos. Podemos comprobarlo aplicando cualquier cambio al componente Trasnform de Escenario, y viendo como afecta a todos los objetos que lo forman [16].

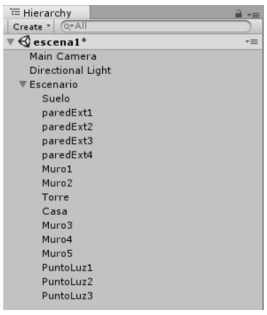


Figura 4.9: Jerarquía de objetos en la escena.

4.5. Incorporando al jugador

El siguiente paso es añadir un jugador que puede moverse libremente por el escenario que hemos creado. El jugador tendrá una vista en primera persona de todo lo que le rodea, pero no podrá verse a sí mismo, por tanto de momento podremos representarlo con un objeto 3D simple. La sombra del jugador si será visible según su posición, así que optaremos por un objeto con forma de cápsula (*GameObject* → *3DObject* → *Capsule*).

Tras añadir el objeto, podemos arrastrar en la jerarquía la cámara de la escena dentro del objeto Capsule, que renombraremos como Jugador. De esta forma es como si viéramos a través de sus ojos. Es importante verificar que la posición de la cámara (en el vector *Position* del componente *Transform*) esté a 0, ya que ahora es una posición relativa al jugador. A pesar de ello puede ser conveniente aumentar la altura de la cámara un poco para que realmente parezca que está situada a la altura de sus ojos (figura 4.10).

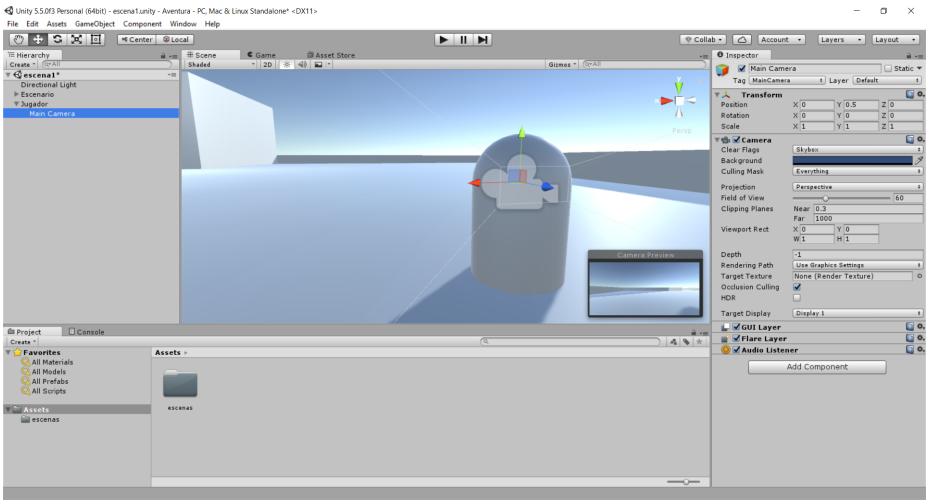


Figura 4.10: Posición de la cámara dentro del jugador.

Si se observa el objeto Jugador en el inspector, se puede observar que tiene asociado un componente *CapsuleCollider*, que Unity incorpora automáticamente. Como su nombre indica, este componente sirve para detectar las colisiones de este objeto con otros, pero en este caso nos interesa sustituirlo por otro componente de tipo *Character Controller*. Esto facilitará más adelante el tratamiento de los controles y la programación del comportamiento de este objeto como un personaje. Para hacerlo, seleccionamos el icono de la rueda dentro del componente *Capsule Collider* y seleccionamos *Remove*

Component. Si pulsamos el botón *Add Component*, aparecen categorías donde se agrupan todo tipo de componentes que podemos añadir al objeto actual, en este caso seleccionamos el componente *Physics* → *Character Controller*.

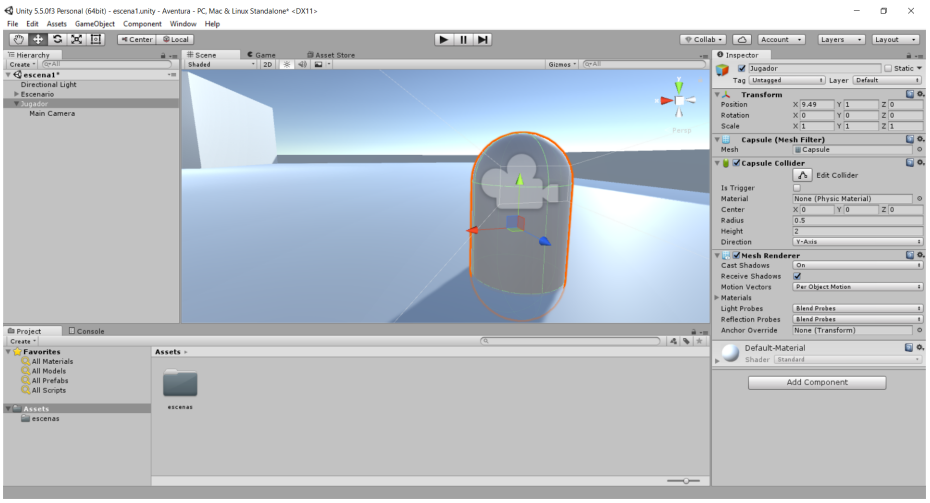


Figura 4.11: Jugador con componente Capsule Collider.

4.5.1. Mirando alrededor

Para que el jugador pueda mirar a su alrededor usando el ratón, tendremos que usar un script que procese la entrada y aplique cambios a los objetos de la escena. Esto es una operación muy común en la metodología de trabajo de Unity, se podría decir que, de forma general, la forma de trabajar consiste en añadir objetos a la escena y asociar scripts a estos objetos para asociarles un comportamiento determinado.

Para la programación de este juego utilizaremos scripts en C#, así que crearemos uno nuevo directorio que llamaremos *scripts* y que aparecerá en la sección de Assets. Para ello se puede seleccionar en el menú *Assets* → *Create* → *Folder*, o accediendo al mismo menú si se pulsa el botón derecho del ratón en la sección *Assets*. A continuación nos situamos dentro del directorio *scripts*, y desde el mismo menú (en *Assets* → *Create* → *C# Script*) podemos crear un nuevo script, al que llamaremos *MirarAlrededor.cs*.

Si seleccionamos el script recién creado, se abrirá el editor que esté configurado por defecto. La instalación de Unity 5.5 incluye dos IDEs preinstalados, Visual Studio 2015 y MonoDevelop (figura 4.12). Para la realización de

este proyecto se utilizará el segundo, aunque la elección no importa demasiado, ya que el IDE se utilizará simplemente para editar el código de los scripts.

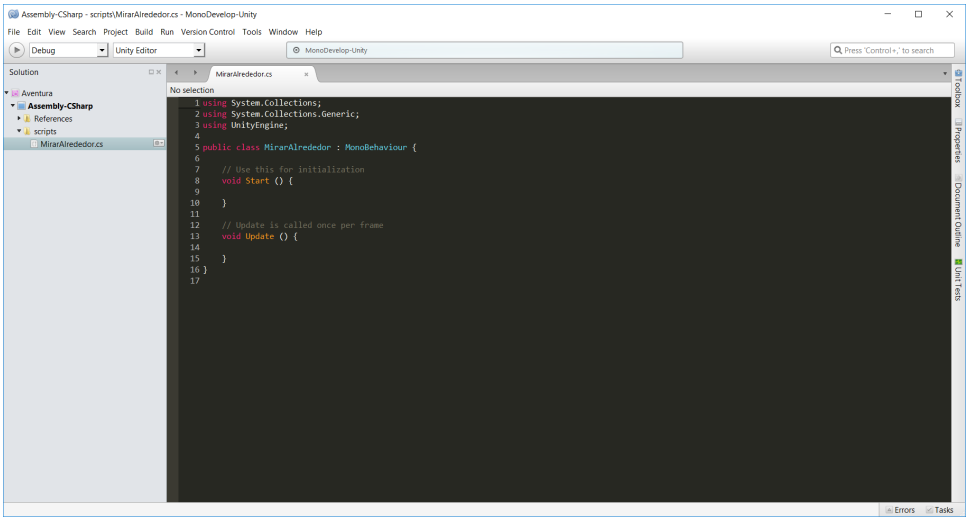


Figura 4.12: Edición de un nuevo script en MonoDevelop.

Tal y como muestra la figura 4.13, en *Edit* → *Preferences* → *External Tools* se puede seleccionar el IDE que Unity abrirá por defecto al editar un script.

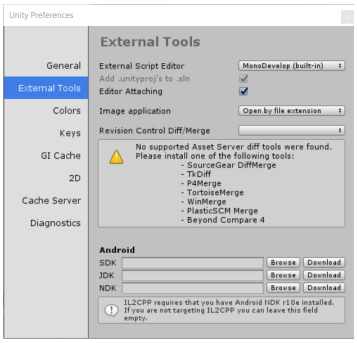


Figura 4.13: Selección de IDE por defecto.

Lo primero que se puede ver al abrir el script *MirandoAlrededor.cs* (figura 4.12) es que ya se ha creado automáticamente una clase del mismo nombre que el fichero, que hereda de la clase *MonoBehaviour* [17]. Todos los scripts de Unity deben heredar de esta clase. Además, se han creado dos métodos,

Start() y *Update()*, que están vacíos. El método *Start()* se llamará una única vez, en el momento en el que el script esté activo. Típicamente aquí se inicializan las variables y cualquier cosa que vaya a ser necesaria durante la ejecución del script.

Como en cualquier videojuego, esta aplicación consistirá en un bucle infinito en el que en cada iteración se procesará la entrada del jugador, se actualizará el estado de la escena y ésta se volverá a dibujar en pantalla. El método *Update()* es un método que se llama en cada una de estas iteraciones, por tanto es el principal método que utilizaremos para actualizar el estado del juego.

Para implementar el movimiento de cabeza del jugador con el ratón, trataremos por separado el movimiento horizontal y el vertical. El siguiente código muestra la componente horizontal:

MirarAlrededor.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MirarAlrededor : MonoBehaviour {

   public enum EjesRotacion {
      Horizontal = 0,
      Vertical = 1,
      Ambos = 2
   }

   public EjesRotacion ejes = EjesRotacion.Ambos;
   public float velocidadRotacion = 12.0f;

   void Start () {

   }

   void Update () {
      // Movimiento horizontal
      if (ejes == EjesRotacion.Horizontal) {
         transform.Rotate(0,
               Input.GetAxis("Mouse X") * velocidadRotacion, 0);
      }
```

```
        else if (ejes == EjesRotacion.Vertical) {
            // Movimiento vertical
        }
        else {
            // Movimiento en ambos ejes
        }
    }
}
```

Este script ya es suficiente para poder probar la rotación horizontal. Puesto que va a afectar al jugador, arrastramos el script desde el directorio correspondiente hasta el objeto del jugador en la vista de jerarquía. En vez de arrastrar el script también es posible vincularlo al objeto Jugador seleccionándolo y añadiendo el script con el botón *Add Component* → *Scripts*.

Se declara una variable de tipo enumeración para poder seleccionar el eje de rotación (horizontal, vertical o ambos, este último será el funcionamiento normal). Definimos también la velocidad de rotación. Declarar estas dos variables como públicas permite ajustar su valor desde el inspector de Unity, incluso en tiempo de ejecución. Esto resulta útil para hacer pruebas, y en este caso seleccionar el eje nos permitirá probar el funcionamiento de la rotación horizontal directamente.

En el método *Update*() especificamos el procesamiento a realizar en cada frame: para el tipo de rotación horizontal basta con usar el método *Rotate()* para rotar el objeto sobre el que está actuando el script (en este caso, el jugador). El método *Rotate()* recibe como parámetro un vector de 3 coordenadas (de tipo *Vector3*).Para rotar horizontalmente sólo actuamos sobre el eje Y, dejando a 0 las otras dos componentes. La detección del movimiento del ratón se consigue con el método *Input.GetAxis()*, que recibe una cadena indicando el eje de movimiento del ratón, que será el eje horizontal. Este valor se multiplica por la velocidad que hayamos definido.

Tras guardar el script, ya podemos pulsar el botón *Play* para ejecutar la aplicación, y podemos comprobar que al mover el ratón horizontalmente también lo hace la visión del jugador. Es importante comprobar en el inspector de la cámara que esté seleccionado el modo de rotación horizontal, y podemos hacer cambios en la velocidad de rotación para obtener la más adecuada.

A continuación implementaremos el movimiento vertical y también la combinación de ambos ejes, para que el jugador pueda mirar alrededor libremente. El siguiente código muestra el script completo:

MirarAlrededor.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MirarAlrededor : MonoBehaviour {

 public enum EjesRotacion {
  Horizontal = 0,
  Vertical = 1,
  Ambos = 2
 }

 public EjesRotacion ejes = EjesRotacion.Ambos;
 public float velocidadRotacionH = 12.0f;
 public float velocidadRotacionV = 12.0f;

 public float minAngleVert = -45.0f;
 public float maxAngleVert = 45.0f;

 // rotación vertical (eje X)
 private float _rotacionX = 0;

 void Start () {
  // Desactiva detección de fisicas para la rotación
  Rigidbody body = GetComponent<Rigidbody>();
  if (body != null) {
   body.freezeRotation = true;
  }
 }


 void Update () {
  // Movimiento horizontal
  if (ejes == EjesRotacion.Horizontal) {
   // Aplica rotación directamente
   transform.Rotate(0,
    Input.GetAxis("Mouse X") * velocidadRotacionH, 0);
  }
  // Movimiento vertical
  else if (ejes == EjesRotacion.Vertical) {
   // Calcula ángulo rotación vertical en frame actual
   _rotacionX -= Input.GetAxis("Mouse Y") * velocidadRotacionV;
```

```
    _rotacionX = Mathf.Clamp(_rotacionX, minAngleVert, maxAngleVert);

    // La rotación horizontal no se incrementa
    float rotacionY = transform.localEulerAngles.y;

    transform.localEulerAngles = new Vector3(_rotacionX, rotacionY, 0);
   }
   // Movimiento en ambos ejes
   else {
    // Calcula ángulo rotación vertical en frame actual
    _rotacionX -= Input.GetAxis("Mouse Y") * velocidadRotacionV;
    _rotacionX = Mathf.Clamp(_rotacionX, minAngleVert, maxAngleVert);

    // Variación rotación horizontal
    float delta = Input.GetAxis("Mouse X") * velocidadRotacionH;
    float rotacionY = transform.localEulerAngles.y + delta;

    transform.localEulerAngles = new Vector3(_rotacionX, rotacionY, 0);
   }
  }
}
```

La implementación del movimiento vertical es algo distinta a la del horizontal, como se puede apreciar en el código. Esto es debido a que en el primer caso se aplicaba directamente la rotación a partir del ratón, pero ahora es necesario limitar el ángulo de giro para evitar que el jugador pueda dar una vuelta completa de arriba a abajo. Por tanto en primer lugar calculamos la variación del ángulo de rotación vertical (respecto al eje X en el sistema de coordenadas de Unity) con *Input.GetAxis("Mouse Y")*, que corresponde al movimiento horizontal del ratón. En cada iteración decrementamos esta cantidad al valor de la iteración anterior. Despúes de calcular el ángulo a aplicar, se utiliza el método *Clamp()* para que este valor no supere el límite de 45º o -45º.

Para aplicar la rotación que hemos calculado, la expresamos como un Vector3 y la asignamos directamente a la propiedad *localEulerAngles* del componente *Transform*. El método *Rotate()* utilizado anteriormente realizaba implícitamente este proceso.

Si el movimiento seleccionado tiene que tener en cuenta ambos ejes, la única diferencia respecto al movimiento vertical es que al calcular el nuevo Vector3, incrementamos la componente correspondiente a la rotación hori-

zontal en un valor delta, que es el valor que obtenemos directamente del movimiento horizontal del ratón.

Por último, en el método *Start()* se habilita el atributo *freezeRotation* del componente *RigidBody* del jugador, si está presente. Esto es por cuestiones de rendimiento, ya que al activarlo no se tienen en cuenta las físicas de los objetos en la rotación del jugador. Para esta aplicación nos interesa que sólo se considere la entrada mediante ratón.

4.5.2. Movimiento del jugador por el escenario

Ahora que el jugador puede mirar alrededor, implementaremos el desplazamiento del propio jugador por el escenario. Esta vez el dispositivo de entrada será el teclado, aunque el código es muy similar al utilizado para detectar el movimiento del ratón. Para ello creamos un nuevo script al que llamaremos *MovimientoJugador.cs* y lo asociaramos al el objeto *Jugador*. El script contiene el siguiente código:

MovimientoJugador.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovimientoJugador : MonoBehaviour {

  public float vel = 8.0f;
  private CharacterController _characterController;
  public float gravedad = -9.8f;

  void Start () {
    _characterController = GetComponent<CharacterController>();
  }

  void Update () {
    // entrada por teclado
    float movX = Input.GetAxis("Horizontal") * vel;
    float movZ = Input.GetAxis("Vertical") * vel;
    Vector3 vectorMov = new Vector3(movX, 0, movZ);

    // no se puede superar vel. maxima en mov. diagonal
    vectorMov = Vector3.ClampMagnitude(vectorMov, vel);
```

```
    vectorMov.y = gravedad;

    vectorMov = vectorMov * Time.deltaTime;
    // conversion a sist. global de coordenadas
    vectorMov = transform.TransformDirection(vectorMov);
    _characterController.Move(vectorMov);
  }
}
```

En el método *Update()* detectamos la pulsación de las teclas con *Input.GetAxis()* de forma similar a como se hizo con el ratón. Como hicimos entonces, usamos un identificador que representa las teclas necesarias, en este caso "Horizontal"detectará las teclas A, D, ← y → y "Vertical"las teclas W, S, ↑ y ↓. Está asociación está establecida así por defecto, ya que son las teclas más utilizadas en juegos en primera persona. De todas formas es posible cambiarlas con el *InputManager*, accesible en *Edit* → *Project Settings* → *Input*. Desde este menú es posible cambiar la configuración de los controles al detalle (figura 4.14).

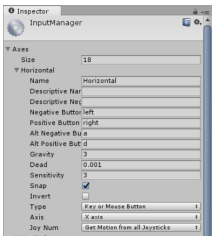


Figura 4.14: Controles.

Una vez obtenidas las componentes vertical y horizontal del movimiento, se mutiplican por una variable para controlar la velocidad del movimiento y se asignan a las componentes X y Z de un nuevo *Vector3*. Después se usa el método *ClampMagnitude* para normalizar la velocidad a la que hemos establecido, ya que si no se hiciera, la velocidad al desplazarse en diagonal sería algo superior a la experimentada al desplazarse sólo vertical u horizontalmente. También hay declarada una variable *gravedad* que asignamos a la componente Y del vector, y que sirve para evitar que el jugador pueda flotar por el escenario.

A continuación hay algunos conceptos que también es importante comentar. En primer lugar hay que tener en cuenta que hasta el momento se ha conseguido implementar el movimiento de los objetos cambiando su posición algunas unidades de distancia en cada iteración. Sin embargo, el número de iteraciones que se ejecuta cada segundo es distinta en cada máquina, dependiendo de su capacidad de procesamiento. Por tanto la velocidad de movimiento de los objetos sería distinta y dependería de la máquina que estuviera ejecutando la aplicación. Para evitar esto, una posible solución es variar la distancia que tienen que desplazarse los objetos en cada iteración para que la velocidad a la que se desplacen sea la misma e independiente de la máquina. Unity proporciona el valor *deltaTime*, disponible en la clase *Time*, que

almacena el tiempo empleado en procesar la última iteración en segundos. Si multiplicamos este valor por la distancia a recorrer en el método *Update()*, conseguimos una velocidad de movimiento independiente de la máquina. Por este motivo multiplicamos el vector de movimiento por *Time.deltaTime*.

Por otra parte, en el entorno de Unity hay dos sistemas de coordenadas distintos: el global y el local de cada objeto particular. Para desplazar al jugador, utilizaremos el método *Move()* del componente *CharacterControler* que le asignamos anteriormente. Este método aplica un movimiento al objeto y recibe como parámetro un vector en el sistema de coordenadas global, así que es necesario convertir el vector del sistema local al global mediante el método *TransformDirection()*. Para poder acceder al componente CharacterController, se usa una variable que se inicializa en *Start()* con el método *GetComponent()*. Este es un método que se utiliza muy a menudo para obtener cualquier componente del objeto (GameObject) al cual está vinculado un script.

Con el script implementado el jugador ya puede moverse por el escenario, aunque hay un detalle que queda por ajustar. La gravedad se aplica sobre la componente Y del movimiento aplicado al jugador, pero si este se inclina hacia arriba o abajo cuando mira en estas direcciones, este vector cambia su dirección y la gravedad le impulsa horizontalmente en vez de hacia abajo.

Una posible solución podría ser asociar el script *MirarAlrededor.cs* a la cámara en vez de al jugador, aunque entonces el desplazamiento del jugador no tendría en cuenta hacia dónde está mirando. En vez de eso, asignaremos este script tanto a la cámara como al jugador, pero estableceremos en el inspector que el jugador sólo rote horizontalmente, y la cámara sólo verticalmente. De esta forma ambos objetos responderán a la entrada desde el ratón por separado: si el ratón se desplaza en horizontal, el jugador rotará horizontalmente, y si se desplaza en vertical, la cámara rotará en vertical, evitando que lo haga el jugador y cambie la dirección de la gravedad.

Con estos elementos ya tenemos un prototipo de videojuego que nos permite recorrer en primera persona el escenario que hemos diseñado. El siguiente paso será añadir otros elementos (como enemigos) con los que se pueda interactuar de varias maneras.

4.6. Implementación de las mecánicas principales

Ahora que es posible explorar el escenario con facilidad puede ser un buen momento para iterar en el diseño del mapa haciendo algunos cambios, ya que la escala del entorno no se percibe igual con una vista general que cuando realmente se recorre en primera persona. Cambiaremos la posición de algunos muros y crearemos algunos nuevos para que se parezca más a un laberinto (figura 4.15), si bien por el momento no dejará de ser un prototipo que nos permita implementar las mecánicas principales del juego.

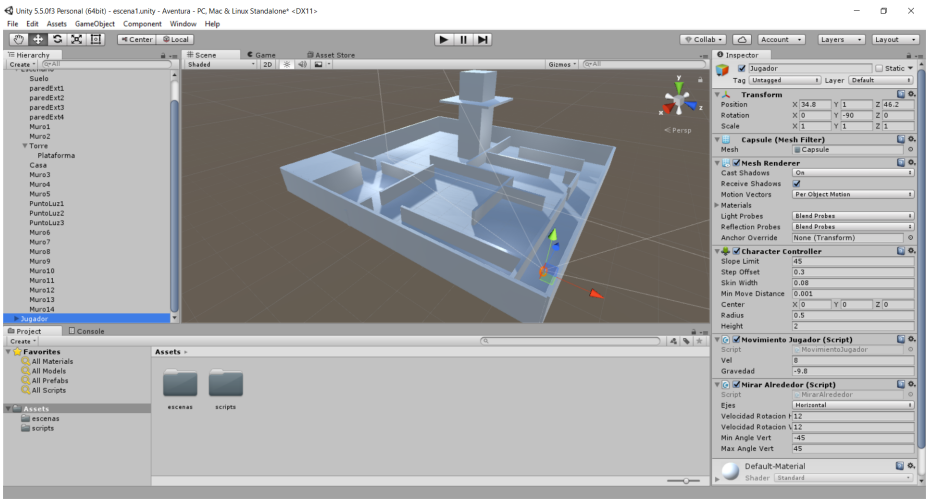


Figura 4.15: Escenario rediseñado.

En cuanto al juego en si, la idea general es que el jugador se mueva por un escenario buscando enemigos y pueda derrotarlos golpeándolos desde corta distancia. Los enemigos se moverán por el mapa, y podrán disparar al jugador si este entra en su campo de visión. Cuando un enemigo sea golpeado, desaparecerá y se regenerará en otro lugar. El objetivo será derrotar a un cierto número de enemigos evitando sus disparos.

4.6.1. Enemigos móviles

Para representar a los enemigos usaremos por el momento formas básicas, de la misma forma que hicimos con el propio jugador. Comenzaremos añadiendo una esfera en cualquier parte del escenario (*GameObject* → *3D Object* → *Sphere*) y la redimensionaremos para que sea más visible y tenga una forma ovalada (figura 4.16).

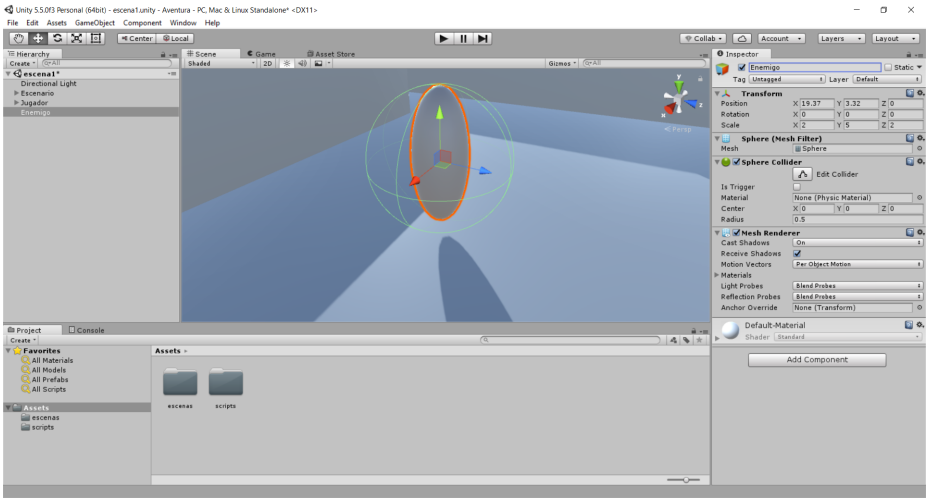


Figura 4.16: Enemigo.

Para que los enemigos se muevan independientemente y su movimiento no sea completamente predecible, tendremos que dotarles de cierta inteligencia artificial. Para ello optaremos por un algoritmo que resulta efectivo en este caso: los enemigos se moverán en linea recta hasta que encuentren un obstáculo en el camino, y antes de colisionar con él cambiarán de dirección (se aplicará sobre ellos una rotación aleatoria).

Para implementar este algoritmo se usará una técnica conocida como *Raycasting*, que se emplea con frecuencia para la detección de colisiones de disparos o proyectiles, pero también se puede emplear para detectar obstáculos en un área cercana. La técnica consiste en proyectar una línea imaginaria a partir de un punto origen hacia una dirección determinada, y detectar la intersección de esta línea con cualquier otro objeto en el camino. Esto normalmente implica realizar cálculos bastante complejos, especialmente en un entorno 3D, sin embargo Unity dispone de clases para facilitar su uso.

A continuación crearemos un script al que llamaremos *IAEnemigo* y lo asociaremos al enemigo recién creado:

IAEnemigo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IAEnemigo : MonoBehaviour {
```

```
  public float vel = 5.0f;
  public float rangoVision = 6.0f;

  void Update () {
    transform.Translate(0, 0, vel * Time.deltaTime);

    Ray ray = new Ray(transform.TransformPoint(
                                    new Vector3(0, 1.5f, 1.5f)),
                                    transform.forward);
    RaycastHit hit;

    if(Physics.Raycast(ray, out hit)) {
      if (hit.distance < rangoVision) {
        float giro = Random.Range(-150, 150);
        transform.Rotate(0, giro, 0);
      }
    }
  }
}
```

En el método *Update()* se desplaza al enemigo a cierta velocidad en línea recta. Con la clase *Ray* se establecen los parámetros del raycast, que comienza en la posición del propio enemigo y avanza en la dirección de lo que este tiene delante (para ello se usa el vector *transform.forward*). Con el método *Raycast()* [18] se proyecta el raycast propiamente dicho. El último parámetro de este método, *hit*, es un parámetro de salida que devuelve información de los obstáculos con los que el rayo ha colisionado. Con esta información se puede verificar si un obstáculo está a cierta distancia, y en ese caso aplicar sobre el eje Y una rotación aleatoria.

Si ejecutamos la aplicación, podemos comprobar cómo el enemigo se mueve y evita chocar con las paredes.

4.6.2. Disparos enemigos y detección de daño

El siguiente paso es hacer que los enemigos disparen al jugador cuando este se acerque a ellos. La implementación podría afrontarse de dos maneras: usando *raycasts* para representar los disparos, en cuyo caso serían instantáneos, o mediante un objeto que represente el proyectil. Optaremos por la segunda opción, ya que resulta interesante que el jugador pueda esquivar los proyectiles cuando se acerque a los enemigos, sin embargo usaremos raycasts

de nuevo para la detección del jugador.

Para incluir los proyectiles, haremos uso de *prefabs*. Un *prefab* no es más que un objeto que se puede guardar en el proyecto como si fuera un *asset* de Unity, con todas sus propiedades y componentes asociados. Hacer uso de *prefabs* no sólo puede resultar cómodo para ahorrar tiempo incluyendo objetos idénticos en la escena, también son necesarios a la hora de instanciar nuevos objetos de forma programática.

Si queremos crear un prefab, arrastramos cualquier objeto de la escena al explorador de *Assets* del proyecto. Crearemos un nuevo directorio *prefabs* para mantener la estructura de ficheros organizada. A continuación creamos un nuevo objeto Sphere y lo renombramos como *Proyectil*. Reduciremos también su tamaño para que lo parezca realmente, y entonces lo arrastramos al directorio *prefabs*. Al hacerlo el objeto correspondiente aparecerá de color azul en la vista de jerarquía (figura 4.17), indicando que está asociado un *prefab*. El objeto en si ya puede eliminarse de la escena, ya que se instanciará usando el *prefab* cada vez que un enemigo dispare.

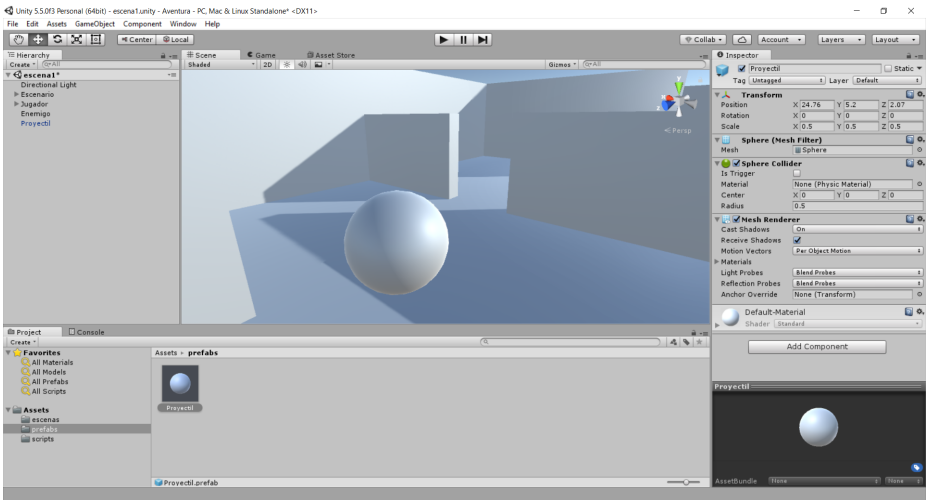


Figura 4.17: Prefab del proyectil.

En caso de que se quisiera editar el prefab, habría que añadirlo a la escena, modificar las propiedades del objeto y después seleccionar *GameObject* → *Apply Changes to Prefab*. Esto lo haremos más adelante, cuando pulamos un poco el aspecto del juego añadiendo texturas y materiales.

Ahora que tenemos el prefab preparado, modificaremos el script *IAEnemigo.cs* para que el enemigo que tenemos en la escena dispare al ver al jugador:

IAEnemigo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IAEnemigo : MonoBehaviour {

  public float vel;
  public float rangoVision;
  public float rangoMinDisparo;

  [SerializeField] private GameObject proyectilPrefab;
  private GameObject _proyectil;

  void Start() {
    vel = 5.0f;
    rangoVision = 6.0f;
    rangoMinDisparo = 2.0f;
  }

  void Update () {
    transform.Translate(0, 0, vel * Time.deltaTime);

    Ray ray = new Ray(transform.TransformPoint(
                                    new Vector3(0, 1.5f, 1.5f)),
                                    transform.forward);
    RaycastHit hit;

    if(Physics.Raycast(ray, out hit)) {
      GameObject objAlcanzado = hit.transform.gameObject;
      // jugador a la vista
      if (objAlcanzado.GetComponent<CharacterController>()
          && hit.distance > rangoMinDisparo) {
        if (_proyectil == null) {
          // se instancia proyectil frente al enemigo
          _proyectil = Instantiate(proyectilPrefab) as GameObject;
          _proyectil.transform.position = transform.TransformPoint
                                        (Vector3.forward * 1.5f);
          _proyectil.transform.rotation = transform.rotation;
```

```
        }
      }
      // otro obstaculo a la vista, ignorando los proyectiles
      else if (objAlcanzado.GetComponent<SphereCollider>() == null
               && hit.distance < rangoVision) {
        float giro = Random.Range(-150, 150);
        transform.Rotate(0, giro, 0);
      }
    }
  }
```

Lo primero que hacemos es declarar atributos para manejar el proyectil, uno para el *prefab* y otro para el propio objeto en el que se instanciará el proyectil. El que corresponde al *prefab* viene precedido por [SerializedField], que fuerza a que Unity lo considere como *serialized*. Por defecto todos los atributos que son públicos se marcan como *serialized*, y por tanto son accesibles desde el inspector. Sin embargo, en este caso nos interesa que se mantenga privado para asegurarnos de que no sea modificado desde otros scripts, y que al mismo tiempo sea accesible desde el inspector para poder definir el *prefab* que queramos utilizar. Esto último es lo que conseguimos con [SerializedField], en cualquier caso, también podría definirse como público y el resultado sería el mismo, con la diferencia de que el atributo sería accesible desde cualquier script.

Mediante la variable *hit* devuelta por el método *Raycast()*, podemos comprobar el objeto que el enemigo tiene a la vista. Si el objeto contiene el componente *PlayerController* se trata del jugador, en ese caso instanciamos el *prefab* que recibe el script mediante el método *Instantiate*, y situamos el proyectil frente al enemigo con su misma rotación. Si no se trata del jugador, se sigue la misma lógica que antes para evitar las paredes, pero esta vez comprobando que el objeto alcanzado no contenga el componente *SphereCollider*. De esta forma evitamos que el enemigo considere al proyectil como un obstáculo y gire cada vez que dispare.

También se ha añadido un método *Start()* para inicializar las variables y una nueva variable, *rangoMinDisparo*, para controlar la distancia mínima a la que tiene que estar el jugador para que el enemigo dispare. Esto sirve para evitar un problema que podría ocurrir cuando el proyectil esté en movimiento, y que se comentará en la implementación del siguiente script.

Antes de probar los cambios, debemos pasarle al script el *prefab* que se quiere instanciar, como se ha mencionado anteriormente. Si se selecciona

el objeto *Enemigo* que tenemos en la escena, se puede observar que en el script asociado hay una nueva casilla a la que debemos arrastrar el *prefab* del proyectil. Por otra parte, también es importante que el enemigo y el jugador estén aproximadamente a la misma altura, ya que en caso contrario el primero no detectaría al segundo, aunque lo tuviera delante. Esto se puede ajustar variando la altura de los objetos, o alternativamente utilizando el método *SphereCast()* para detectar un área más amplia frente al jugador. En la figura 4.18 se puede ver como los dos objetos están a una altura similar, y cómo hemos asignado el prefab del proyectil al script *IAEnemigo*.

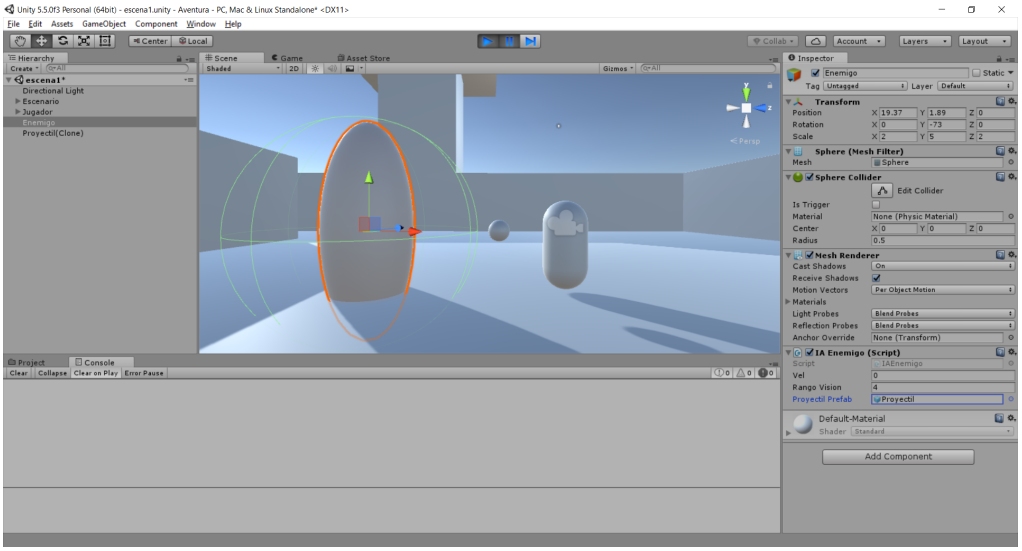


Figura 4.18: Enemigo disparando al jugador.

Si ejecutamos la aplicación, al situarnos enfrente del enemigo aparecerá el proyectil, aunque estará detenido. Tendremos que asociar al prefab su propio script para ponerlo en movimiento y detectar su colisión con el jugador. El código de este nuevo script, *ControlProyectil.cs*, es el siguiente:

ControlProyectil.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControlProyectil : MonoBehaviour {

   public float vel = 5.0f;
   public int fuerza = 1;
```

```
    void Update () {
        transform.Translate(0, 0, vel * Time.deltaTime);
    }

    void OnTriggerEnter(Collider other) {
        ControlJugador jug = other.GetComponent<ControlJugador>();
        if (jug != null) {
            jug.RecibirDisparo(fuerza);
        }
        Destroy(this.gameObject);
    }
}
```

En *Update()* se actualiza la posición del proyectil para que se mueva hacia adelante. Por otra parte tenemos *OnTriggerEnter()*, que es un método predefinido en Unity donde especificamos el código que se ejecutará cuando el proyectil colisione con otro objeto (concretamente se detecta la colisión con el *Collider* de otro objeto, que es lo que recibe por parámetro). A partir del *Collider* comprobamos si el objeto impactado es el jugador. Esto se puede averiguar comprobando si el objeto tiene asignado el script *ControlJugador.cs*, que crearemos a continuación. Si el jugador es alcanzado, llamamos a un método que añadiremos a *ControlJugador* y que mostrará por la consola de Unity un mensaje. En cualquier caso, cuando el proyectil colisione con otro objeto lo eliminamos de la escena con el método *Destroy()*.

La variable *rangoMinDisparo* que añadimos en *IAEnemigo.cs*, y que establecía una distancia mínima entre el enemigo y el jugador para poder disparar, evita un problema que se puede producir si el jugador se sitúa muy cerca del enemigo y justo enfrente: en ese caso el proyectil recorre una distancia muy corta antes de destruirse, por tanto el enemigo podría disparar repetidamente en un intervalo muy corto de tiempo. Esto también se podría solucionar introduciendo un pequeño retardo entre cada disparo.

Después de asignar el script al prefab del proyectil, es necesario marcar la casilla *Is Trigger* en el componente *SphereCollider* del prefab y añadir un componente *RigidBody* (*Add Component* → *Physics* → *RigidBody*) para que puedan detectarse las colisiones. También desmarcamos la casilla *Use Gravity* en este componente para que al proyectil no le afecte la gravedad.

Para poder ejecutar la aplicación todavía hace falta un paso más: crear el script *ControlJugador.cs* al que se hace referencia en el script del proyectil:

ControlJugador.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControlJugador : MonoBehaviour {

   private int _vidas;

   void Start () {
      _vidas = 10;
   }

   public void RecibirDisparo(int fuerza) {
      _vidas -= fuerza;
      Debug.Log("Vidas: " + _vidas);
   }
}
```

En este script simplemente declaramos el número de vidas que tiene el jugador y un método que las decrementa al recibir un disparo y las muestra por la consola. Tras asignar el script al jugador, finalmente se puede probar el juego y comprobar que el proyectil se mueve y se detectan las colisiones correctamente.

4.6.3. Golpeando enemigos

En este apartado implementaremos el ataque del jugador a los enemigos, que se realizará a corta distancia golpeándoles con un bastón. Por el momento lo representaremos añadiendo un cilindro en la escena mediante *GameObject* → *3D Object* → *Cylinder* y cambiando su tamaño y posición para que tenga forma alargada y quede situado justo enfrente de la cámara (figura 4.19). También haremos que el bastón sea un hijo del jugador en la jerarquía de objetos, al igual que se hizo con la propia cámara. De está forma, sus coordenadas serán relativas al jugador y el bastón lo seguirá cuando se mueva por la escena. Para evitar posibles problemas con las colisiones de los proyectiles enemigos, eliminamos el componente *CapsuleCollider*.

A continuación queremos que el jugador pueda golpear a los enemigos cuando esté cerca de ellos. También se tendrá que comprobar que esté mirando a su objetivo, ya que no sería muy realista si se detectara el golpe

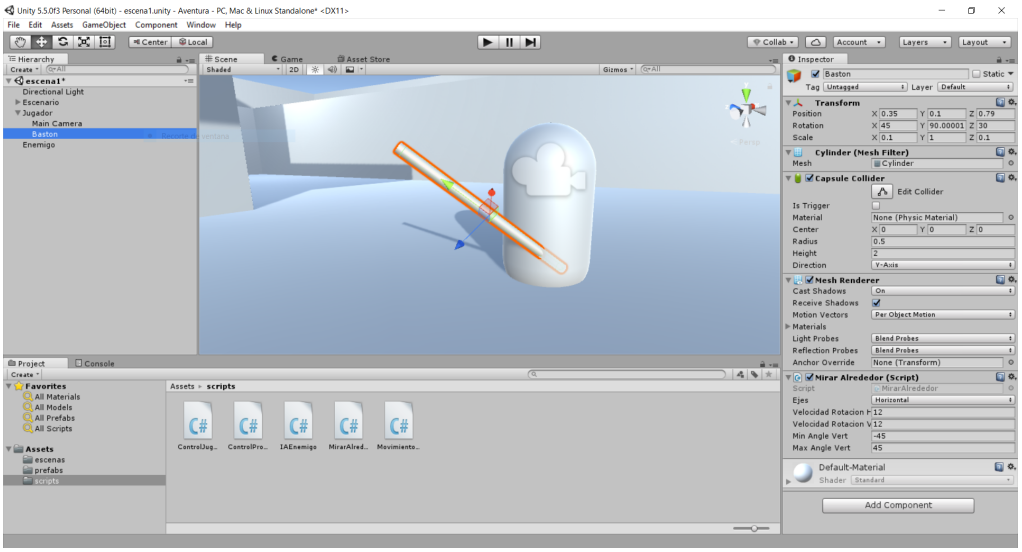


Figura 4.19: Bastón del jugador.

cuando el jugador estuviera de espaldas, por ejemplo. Para ello modificaremos el script *ControlJugador.cs* añadiendo un método *Update()*, y variables para llevar la cuenta de los puntos conseguidos y los puntos máximos (necesarios para ganar una partida):

ControlJugador.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControlJugador : MonoBehaviour {

  private int _vidas;
  private int _puntos;
  private int _rangoAtaque;
  private int _maxPuntos;

  private Camera _camara;

  void Start () {
    _vidas = 3;
    _puntos = 0;
    _maxPuntos = 15;
    _rangoAtaque = 1;
```

```
    _camara = GetComponentInChildren<Camera>();
  }

  void Update() {

    // boton izq raton pulsado
    if (Input.GetMouseButtonDown(0)) {

      // se emite raycast hacia el centro de la cámara
      Vector3 centroCamara = new Vector3(_camara.pixelWidth / 2,
                                         _camara.pixelHeight / 2,
                                         0);
      Ray ray = _camara.ScreenPointToRay(centroCamara);
      RaycastHit hit;

      if (Physics.Raycast(ray, out hit)) {

        GameObject objetoAlcanzado = hit.transform.gameObject;
        IAEnemigo scriptEnem =
                    objetoAlcanzado.GetComponent<IAEnemigo>();

        // si es un enemigo, está dentro del rango y está vivo
        // recibe golpe
        if (scriptEnem != null && hit.distance < _rangoAtaque
            && scriptEnem.vidas > 0) {
          scriptEnem.RecibirGolpe();
          _puntos++;
          Debug.Log("Puntos: " + _puntos);
        }
      }
    }

  }

  public void RecibirDisparo(int fuerza) {
    _vidas -= fuerza;
    Debug.Log("Vidas: " + _vidas);
  }

}
```

En el script se vuelve a hacer uso de raycasts para la detección del golpe. En el método *Start()* inicializamos el rango en el que queremos detectar al enemigo y obtenemos la cámara principal, ya que tendremos que hacer

referencia a ella en *Update()*. Puesto que anteriormente hicimos que el objeto de la cámara fuera hijo del jugador, utilizamos *GetComponentInChildren<Camera>()*.

En el propio método *Update()*, detectamos si se ha pulsado el botón izquierdo del ratón mediante Input.GetMouseButtonDown(0). En caso afirmativo, se proyecta un raycast desde la posición de la cámara, que se dirige hacia el centro de la pantalla, con el método *ScreenPointToRay()*. El código que obtiene la información del objeto alcanzado es similar al explicado anteriormente, con una diferencia importante: en este caso se utiliza el script *IAEnemigo.cs* para saber si el objeto es en efecto un enemigo, y además se utiliza este script para llamar al método *RecibirGolpe()* en caso de que esté dentro del rango de ataque. Este método tenemos que añadirlo a *IAEnemigo.cs*:

IAEnemigo.cs

```
public int vidas;

void Start() {
   vidas = 1;
}

void Update() {
   [...]
}

public void RecibirGolpe() {
   vidas--;
   if (vidas < 1) {
      StartCoroutine(Desaparecer());
   }
}

private IEnumerator Desaparecer() {
   vel = 0;
   transform.Translate(0, -1, 0);
   yield return new WaitForSeconds(0.5f);
   transform.Translate(0, -1, 0);
   yield return new WaitForSeconds(0.5f);
   transform.Translate(0, -1, 0);
   yield return new WaitForSeconds(0.5f);
   Destroy(this.gameObject);
```

```
}
```

Como se puede ver, se añade una variable *vidas* para controlar el número de golpes que tienen que recibir los enemigos para desaparecer. En cuanto al método *RecibirGolpe()*, este decrementa el número de vidas del enemigo y si se agotan llama a una co-rutina.

Una co-rutina (*Coroutine*) es básicamente un método especial que puede pausarse sin detener el resto de la aplicación, lo cual nos puede resultar útil para que los enemigos muestren una pequeña animación al desaparecer. Con *StartCoroutine()* se inicia la co-rutina *Desaparecer()*, que mueve al enemigo hacia abajo y se pausa durante unos instantes repetidamente, hasta que queda completamente por debajo del suelo. Cuando esto ocurre, destruimos el objeto *Enemigo* con *Destroy()*. Esta es una posibilidad a la hora de implementar pequeñas animaciones. Unity dispone de otros sistemas para usar animaciones más elaboradas, que se pueden considerar a la hora de pulir el aspecto y la calidad gráfica en general.

Con todos estos cambios el jugador ya puede golpear a los enemigos, aunque el bastón seguirá sin moverse. Con un script, *MovimientoBaston.cs*, podemos cambiar la posición del bastón cada vez que el jugador pulse o suelte el botón izquierdo del ratón:

MovimientoBaston.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovimientoBaston : MonoBehaviour {

   void Update () {

      if (Input.GetMouseButtonDown(0)) {
         // Posicion por defecto
         transform.localRotation = Quaternion.Euler(35, 90, 35);
      }

      if (Input.GetMouseButtonUp(0)) {
         // Golpe
         transform.localRotation = Quaternion.Euler(35, 90, 15);
```

```
      }
   }
}
```

Para que el movimiento aplicado sea el esperado, independientemente de la orientación del bastón en la escena, se modifica la rotación local del objeto asignándola directamente a transform.localRotation. Puesto que la asignación es directa, debemos usar *Quarternion.Euler()* para que se haga la conversión al sistema que usa Unity para aplicar las rotaciones.

4.6.4. Regeneración de enemigos

Para probar los cambios hechos hasta el momento hemos tenido a un único enemigo en la escena, en este apartado añadiremos varios para que el jugador tenga que buscarlos por el escenario. Aunque se añadieran muchos, todo enemigo derrotado no vuelve a aparecer, y puede llegar un momento en el que ya no queden más enemigos. En vez de eso, pretendemos que haya en todo momento un número predeterminado de enemigos, y que el juego termine cuando se haya derrotado a cierto número de ellos o el jugador pierda todas sus vidas.

Para lograr este comportamiento, es necesario llevar la cuenta de los enemigos que están presentes, y regenerar a uno cada vez que sea eliminado. Por tanto crearemos un objeto vacío, al que llamaremos *Regenerador*, que se encargará de esta tarea. El objeto estará vacío porque se trata de un controlador abstracto, que no será visible en ningún momento, pero que implementará parte de la lógica del juego. Mediante *GameObject* → *Create Empty* creamos el objeto vacío, y lo colocamos en cualquier parte de la escena (su posición no importa, por simplicidad se puede colocar en el origen) y le asignamos un nuevo script, *RegenerarEnemigo.cs*:

RegenerarEnemigo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RegenerarEnemigo : MonoBehaviour {

 [SerializeField] private GameObject prefab;
 private GameObject _enemigo;
```

```
 public void Regenera() {
  _enemigo = Instantiate(prefab) as GameObject;

  int pos = Random.Range(0, 3);
  switch (pos) {
   case 0:
    _enemigo.transform.position = new Vector3(0f, 0.8f, -30f);
    break;

   case 1:
    _enemigo.transform.position = new Vector3(30f, 0.8f, -20f);
    break;

   case 2:
    _enemigo.transform.position = new Vector3(20f, 0.8f, 0f);
    break;

   default:
    _enemigo.transform.position = new Vector3(-15f, 0.8f, -30f);
    break;
  }

  float angulo = Random.Range(0, 360);
  _enemigo.transform.Rotate(0, angulo, 0);
 }
}
```

Este script es similar al que se utilizó para instanciar los proyectiles que disparan los enemigos. Para que funcione correctamente, de la misma forma que se hizo con el proyectil, hay que crear un *prefab* para los enemigos (a partir de cualquier *GameObject* Enemigo) y arrastrarlo a la casilla de este nuevo script. El código define cuatro posiciones predefinidas repartidas por el escenario. Cuando se llama a la función *Regenera()*, se elige aleatoriamente una de estas posiciones y se instancia un nuevo enemigo en ella, con una rotación también aleatoria. De esta forma el jugador no podrá predecir dónde encontrará a los enemigos según se vayan regenerando. Sólo hace falta añadir la llamada a esta función desde el script *IAEnemigo.cs*, en el método *RecibirGolpe()*:

IAEnemigo.cs

```
public void RecibirGolpe() {
```

```
   vidas--;
   if (vidas < 1) {
      StartCoroutine(Desaparecer());

      //se llama al script que regenera un enemigo nuevo
      GameObject regenerador = GameObject.Find("Regenerador");
      RegenerarEnemigo scriptReg =
          regenerador.GetComponent<RegenerarEnemigo>();
         scriptReg.Regenera();
   }
}
```

Aprovechando que tenemos creado el prefab para los enemigos, añadiremos algunos más para que el jugador pueda encontrarse a varios desde el inicio del juego. Con estos cambios, podemos ejecutarlo y comprobar que al derrotar a un enemigo aparece uno nuevo en otro lugar aleatorio. El número de enemigos se mantendrá constante.

4.7. Interfaz de usuario (HUD)

Para terminar con la funcionalidad del juego, resulta conveniente tener indicadores en pantalla para mostrar las vidas del jugador y los puntos conseguidos. En Unity existen principalmente dos formas de implementar una interfaz visual (o HUD): usando el sistema tradicional, disponible desde la primera versión del motor, o el nuevo sistema.

El primero, también conocido como modo inmediato, define todos los elementos únicamente mediante código, aunque en cuanto a funcionalidad está algo limitado en comparación con el modo avanzado, que el más reciente. En este caso usaremos este último, ya que a pesar de que el HUD que vamos a implementar es bastante minimalista, ofrece ciertas ventajas. El nuevo sistema permite definir y manipular los elementos de la interfaz desde el propio editor, además de que resulta más flexible a la hora de incluir elementos más complejos que sólo texto, como por ejemplo imágenes o controles de todo tipo.

Para comenzar, añadimos un objeto de tipo Canvas con *GameObject* → *UI* → *Canvas*, y lo renombramos como *HUD*. Unity añadirá además un objeto *EventSystem* de forma automática, que está relacionado con el manejo de eventos, aunque para lo que pretendemos implementar no será necesario usarlo. Al añadir el HUD aparecerá un recuadro que representará la pantalla,

y que servirá como lienzo para ir añadiendo los elementos que se quieran incluir (figura 4.20). Este recuadro será muy grande en relación a la escena, ya que un píxel en el Canvas equivale a una unidad de distancia. Para manejarlo con más comodidad, cambiamos al modo de vista en 2D pulsando el botón *2D* en la parte superior de la vista de la escena.

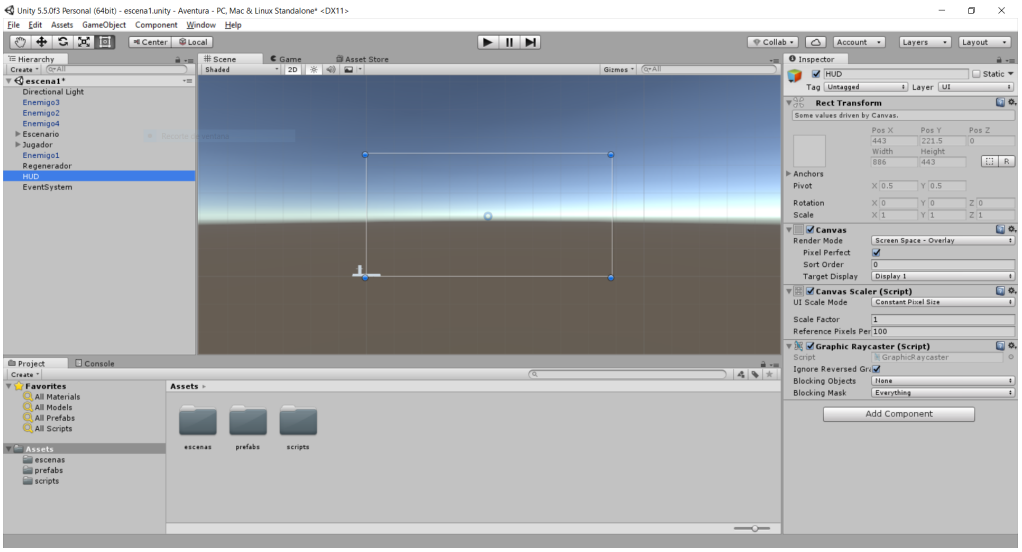


Figura 4.20: Objeto Canvas añadido a la escena.

Dentro del menú *GameObject* → *UI* tenemos disponibles los elementos que podemos utilizar, y que tendrán que añadirse como hijos del Canvas. En nuestro caso, haremos uso de dos objetos de tipo *Text* y dos de tipo *Image*. Una vez situados, los elementos se pueden reposicionar como cualquier otro objeto, y podemos modificar sus propiedades desde el inspector. Una de las propiedades más importantes es el Anchor de cada elemento, representado con un aspa en el HUD. Esta propiedad sirve para fijar la posición de cualquier elemento en relación al HUD, de forma que su posición relativa no cambie aunque el HUD varíe su tamaño cuando se cambie entre resoluciones.

Colocaremos uno de los textos y una imagen en la esquina inferior izquierda para mostrar las vidas del jugador, y haremos lo mismo para los puntos en la esquina inferior derecha. Para los elementos de texto, aunque se vayan a modificar mediante código, podemos asignarles el texto *Vidas* y *Puntos* respectivamente, para poder ver el resultado aproximado desde el editor. En cuanto a los objetos de tipo Image, además de posicionarlos, habrá que importar una imagen como *Sprite* para cada uno de ellos. Tras obtener las

imágenes [19], creamos un nuevo directorio, *sprites*, para almacenar los que necesitemos. Para importar una imagen, basta con arrastrarla desde cualquier explorador al nuevo directorio, o bien usando el menú *Asset* → *Import New Asset*. Además será necesario asegurarse en el inspector de cada sprite que el atributo *Texture Type* esté establecido como *Sprite (2D and UI)*. Con los sprites importados, podemos arrastralos al inspector de los objetos Image en el HUD, en la casilla *Source Image*. De esta forma, ya tendremos listos los iconos y los textos para poder acceder a ellos desde un script que muestre los valores actualizados (figura 4.21).

Figura 4.21: Colocación de los elementos del HUD mediante *Anchors*.

El script que mantendrá actualizados los contadores del HUD estará asignado a un objeto que crearemos vacío. Creamos este objeto (*GameObject* → *Create Empty*) y lo renombramos *ControladorHUD*. A continuación creamos el nuevo script, *ControlHUD.cs*, y lo asignamos al objeto. El código de *ControlHUD.cs* es el siguiente:

ControlHUD.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ControlHUD : MonoBehaviour {
```

```
   [SerializeField] private Text textoVidas;
   [SerializeField] private Text textoPuntos;

   public void ActualizarVidas(int vidas) {
      textoVidas.text = vidas.ToString();
   }



   public void ActualizarPuntos(int puntos, int maxPuntos) {
      textoPuntos.text = puntos.ToString()
                         + "/" + maxPuntos.ToString();
   }
}
```

Como se puede observar, el script recibe los dos objetos *Text* del HUD que muestran las vidas y los puntos como parámetros, por lo que hay que enlazarlos arrastrando estos elementos en el inspector a sus casillas correspondientes. Por lo demás, simplemente se declaran dos métodos que reciben los valores y actualizan los textos en el HUD. La llamada a estos métodos se hará desde *ControlJugador.cs*, en el momento en que cambie el valor de las vidas o de los puntos. Por lo tanto habrá que modificar el script para acceder al controlador del HUD y actualizarlo cuando se reciba un disparo o se golpee a un enemigo:

ControlJugador.cs

```
void Start () {
  [...]
  _controladorHUD = GameObject.Find("ControladorHUD");
  _scriptControlHUD = _controladorHUD.GetComponent<ControlHUD>();
  _scriptControlHUD.ActualizarVidas(_vidas);
  _scriptControlHUD.ActualizarPuntos(_puntos, _maxPuntos);
}

void Update() {
  [...]
  // si es un enemigo, está dentro del rango y está vivo
  // recibe golpe
  if (scriptEnem != null && hit.distance < _rangoAtaque
      && scriptEnem.vidas > 0) {
```

```
    scriptEnem.RecibirGolpe();
    _puntos++;
    _scriptControlHUD.ActualizarPuntos(_puntos, _maxPuntos);
  }
  [...]
}

public void RecibirDisparo(int fuerza) {
  _vidas -= fuerza;
  _scriptControlHUD.ActualizarVidas(_vidas);
}
```

Si ejecutamos el juego, el HUD será visible y se actualizarán los contadores al recibir disparos y al golpear a los enemigos (figura 4.22).

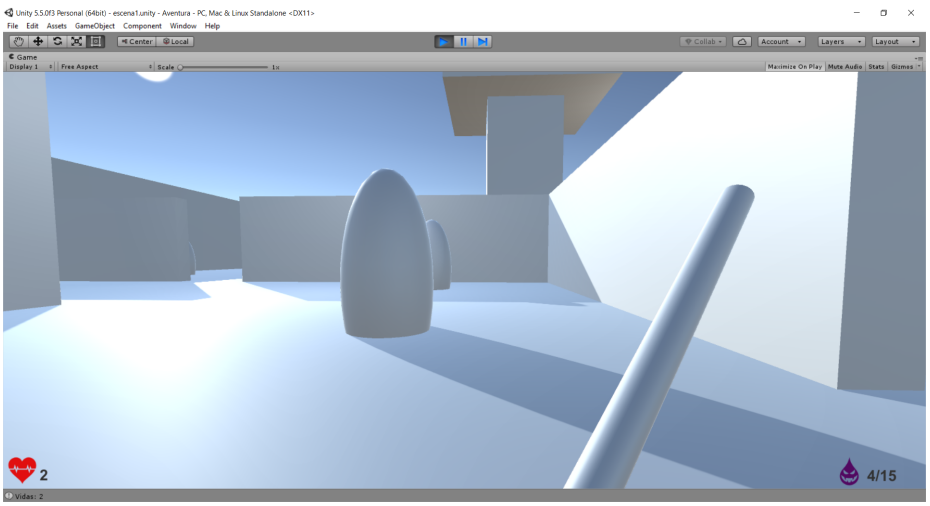


Figura 4.22: Escena con indicadores de vidas y puntos.

4.7.1. Pantallas de victoria y fin de juego

Lo único que le falta al juego para tener completa su funcionalidad es que la partida termine, bien porque el jugador ha derrotado a un cierto número de enemigos o porque ha perdido todas sus vidas. En caso de que esto ocurra, se mostrará una ventana informando de la victoria o derrota, y a continuación la partida comenzará desde el principio para que el jugador lo vuelva a intentar.

Estas dos ventanas se incorporarán a la interfaz visual existente como dos imágenes más, de la misma forma que se hizo con los iconos. Las añadimos

desde el menú *GameObject* → *UI* → *Image* como objetos hijos del HUD, y a cada una le asignamos un sprite que tendremos que importar en Unity como se hizo anteriormente, arrastrando una imagen al directorio *sprites*, cambiando el ajuste *Texture Type* a *Sprite(2D and UI)* y arrastrando el sprite a la casilla *source image* de cada objeto creado (figura 4.23).

Figura 4.23: Pantalla de victoria.

Una vez tengamos los objetos *PantallaVictoria* y *PantallaGameOver* con sus sprites asignados, crearemos un nuevo script, *ControlVentana.cs*, y se lo asignaremos a ambos. Este script simplemente nos permitirá mostrar u ocultar las ventanas.

ControlVentana.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControlVentana : MonoBehaviour {

   public void Abrir() {
      gameObject.SetActive(true);
   }

   public void Cerrar() {
```

```
      gameObject.SetActive(false);
   }
}
```

A continuación añadiremos el siguiente código al script *ControlHUD.cs* para que el HUD pueda acceder a las nuevas ventanas:

ControlHUD.cs

```
public class ControlHUD : MonoBehaviour {

[...]

private GameObject _pantallaVictoria, _pantallaGameOver;
private ControlVentana _scriptVictoria, _scriptGameOver;

void Start() {
  _pantallaVictoria = GameObject.Find("PantallaVictoria");
  _scriptVictoria = _pantallaVictoria.GetComponent<ControlVentana>();
  _scriptVictoria.Cerrar();

  _pantallaGameOver = GameObject.Find("PantallaGameOver");
  _scriptGameOver = _pantallaGameOver.GetComponent<ControlVentana>();
  _scriptGameOver.Cerrar();
}

public void FinPartida(bool victoria) {

  StartCoroutine(CorutinaFinPartida(victoria));
}

private IEnumerator CorutinaFinPartida(bool victoria) {
  if (victoria) {
    _scriptVictoria.Abrir();
  }
  else {
    _scriptGameOver.Abrir();
  }
  yield return new WaitForSeconds(3.0f);
  SceneManager.LoadScene("escena1");
}

[...]
```

```
}
```

En el método *Start()* se accede a los scripts de las pantallas de victoria y derrota, y estas se deshabilitan inicialmente. El método *FinPartida()* se podrá llamar desde otros scripts, y recibe un un valor booleano que indica si la partida ha acabado con victoria o derrota del jugador. Después llama a una co-rutina que es la que realmente ejecuta el código necesario: habilita una ventana u otra según si el jugador ha ganado o perdido, y después de una espera de unos pocos segundos se carga la escena de nuevo con *SceneManager.LoadScene()*. Este método carga la escena completa con sus valores iniciales, por tanto la partida se reinicia.

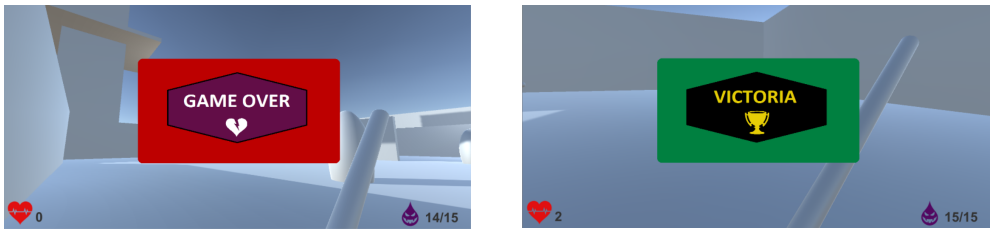
El último paso es realizar la llamada a *FinPartida()* desde *ControlJugador.cs* cuando se pierdan todas las vidas o se alcancen los puntos máximos:

ControlJugador.cs

```
[...]
void Update() {
   [...]
   if (_puntos >= _maxPuntos) {
      _scriptControlHUD.FinPartida(true);
   }
   [...]
}

public void RecibirDisparo(int fuerza) {
   [...]
   if (_vidas < 1) {
      _scriptControlHUD.FinPartida(false);
   }
   [...]
}
```

En este momento podemos ejecutar la aplicación y comprobar que las ventanas aparecen correctamente al perder todas las vidas o llegar al número de puntos objetivo, como muestran las figuras 4.24a y 4.24b.



(a) (b)

Figura 4.24: Pantalla de victoria (a) y de derrota (b).

También añadiremos una pantalla de inicio que aparecerá al lanzar la aplicación. Esta pantalla mantendrá el juego pausado, y pulsar cualquier tecla la ocultará y dará comienzo a la partida. Para ello la añadiremos al HUD de la misma forma que las otras pantallas, y modificaremos *ControlHUD.cs*:

ControlHUD.cs

```
[...]
private GameObject _pantallaStart;

   void Start() {
      [...]
      _pantallaStart = GameObject.Find("PantallaStart");

      // juego en pausa
      Time.timeScale = 0f;

   }

   void Update() {
      // comienza la partida
      if (_pantallaStart.activeInHierarchy && Input.anyKeyDown) {
         _pantallaStart.SetActive(false);
         Time.timeScale = 1f;
      }
   }
```

En *Start()* obtenemos una referencia a la pantalla de inicio y ponemos el juego en pausa poniendo el atributo *Time.timeScale* a 0. En el método *Update()*, verificamos si se ha pulsado cualquier tecla, en cuyo caso deshabilitamos la pantalla y reanudamos el juego.

Con las características que se han implementado hasta el momento podemos decir que el juego ya tiene todos los elementos básicos necesarios para considerarse terminado en cuanto a su funcionalidad, si bien quedarían pendientes otros aspectos como ajustar la dificultad o pulir el aspecto gráfico con modelos más detallados, texturas, etc.

Capítulo 5

Mejora del aspecto gráfico y efectos de sonido

Hasta este punto se han implementado los componentes del juego usando los modelos y formas básicas que proporciona Unity, dejando de lado el apartado gráfico y sonoro. En este capítulo abordaremos cómo mejorar el aspecto general, sustituyendo parte de los modelos existentes por otros más elaborados, añadiendo texturas, sonidos y otros efectos.

Por otra parte, también conviene considerar que si se pretende ejecutar la aplicación en múltiples plataformas, es recomendable no excederse con el número de polígonos, fuentes de luz o cualquier elemento que pueda tener un gran impacto en el rendimiento, sobretodo pensando en dispositivos móviles.

5.1. Materiales

Comenzaremos dando color a algunos elementos mediante materiales. Un material en Unity es un elemento que contiene información acerca del aspecto de un objeto, como su color o luminosidad. Una vez creado puede aplicarse sobre la superficie de cualquier objeto de la escena. Para asignar un color al suelo, por ejemplo, seleccionamos *Assets* → *Create* → *Material* para crear un material. Crearemos también un nuevo directorio *materiales* para guardar en él todos los que necesitemos. En el inspector podemos ver todas las propiedades del material que se pueden definir. Una de las más importantes es *Albedo*, que utilizaremos para definir el color verde del suelo (figura 5.1).

Para aplicar el material al suelo, basta con arrastrarlo al objeto correspondiente, bien en la lista de objetos en la vista de jerarquía o directamente

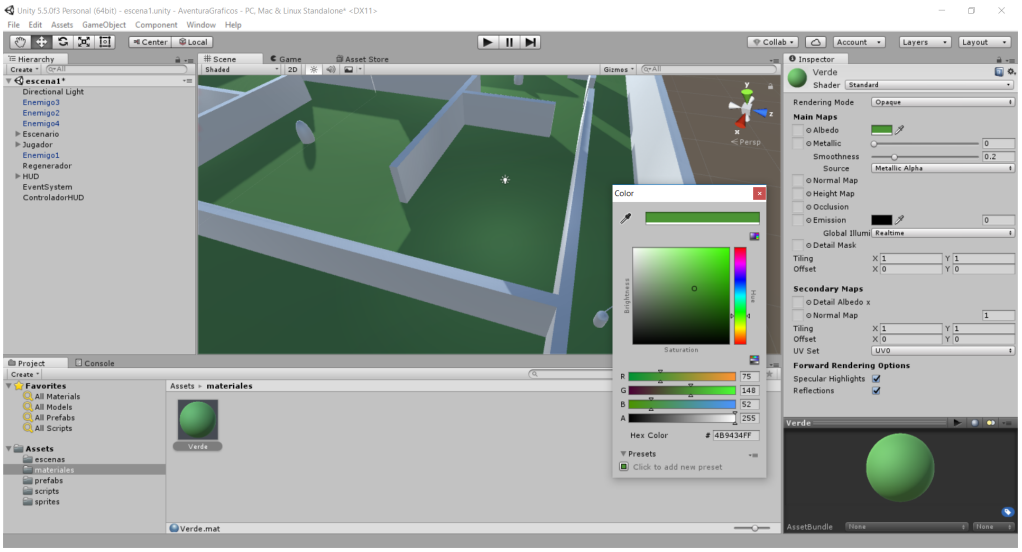


Figura 5.1: Selección de albedo de material.

sobre el objeto en la escena.

Seguiremos un procedimiento similar para asignarle un material al proyectil que disparan los enemigos. Este es un caso especial, ya que se trata de un prefab que se instancia con el juego en ejecución y por tanto no lo tenemos en la escena. Para asignarle el material, debemos añadirlo a propósito para realizar el cambio. En primer lugar creamos un material nuevo, de color amarillo, y establecemos el valor de *Emission* a 0.1 para que brille con algo de luz propia. Después añadimos el prefab a la escena y le asignamos el material, tras lo cual aplicamos los cambios con *GameObject* → *Apply Changes To Prefab*. Con los cambios aplicados, la instancia del objeto ya no es necesaria y puede eliminarse de la escena.

5.2. Texturas basadas en sprites

Además de crear materiales a partir de colores y otras propiedades, también es posible aplicar una textura a un objeto de la escena. Para ello basta con importar una imagen como sprite, como se hizo en apartados anteriores, y crear un nuevo material a partir de ella. A menudo lo más conveniente es que la imagen sea repetible para poder usarla como un *tile*, es decir, que podamos colocar la misma imagen repetidamente y no se perciba la separación de una con la siguiente. De esta forma se pueden crear grandes superficies a partir de una pequeña imagen (figura 5.2).

Figura 5.2: Imagen empleada como tile.

Utilizaremos esta técnica para que las paredes del escenario tengan una textura de tablones de madera. Tras importar la imagen como sprite [20], creamos un nuevo material y arrastramos el sprite a la propiedad Albedo en el inspector. También ponemos la propiedad *Smoothness* a 0 para que el material tenga un aspecto mate. Los parámetros *Tiling* (X e Y) controlan el número de repeticiones de la imagen en ambos ejes. Según cada caso puede ser necesario modificar estos valores, en cambio con el valor de 1 por defecto, no hay repeticiones y el sprite se adapta a la superficie del objeto. El resultado de aplicar las texturas se puede apreciar en la figura 5.3.

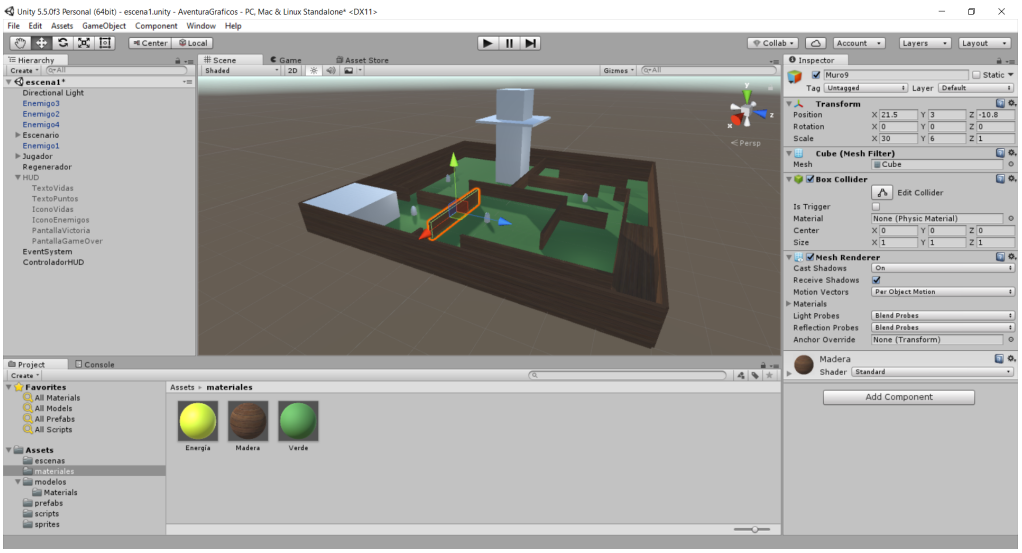


Figura 5.3: Escenario con texturas aplicadas a las paredes.

5.2.1. Texturas del cielo

También se pueden usar texturas para darle al cielo un aspecto más realista que el que Unity proporciona por defecto. Esto se consigue habitualmente creando lo que se conoce como un *Skybox*, que es una caja que rodea la escena

y que tiene una textura de fondo en cada una de sus seis caras.

Para llevarlo a cabo, comenzaremos obteniendo imágenes ya preparadas para usarse como Skybox [21]. Después de importar las imágenes como sprites en Unity, accedemos al inspector en cada una de ellas, y cambiamos el parámetro *Wrap Mode* a *Clamp*. Esto evitará que las líneas de intersección entre las caras del cubo sean visibles. Para que el cielo de la sensación de ser uniforme y de estar muy alejado de la escena, se usará un material con un *shader* especial. En todos los materiales usados hasta el momento se ha empleado el shader *Standard* que viene establecido por defecto. En esta ocasión crearemos un material al que llamaremos Cielo, y cambiaremos el shader a *Skybox/6-sided*. El inspector del material cambiará para mostrar 6 casillas en las que tendremos que introducir las imágenes importadas. Por último, asignaremos este material al skybox de la escena en el menú *Window* → *Lighting*.

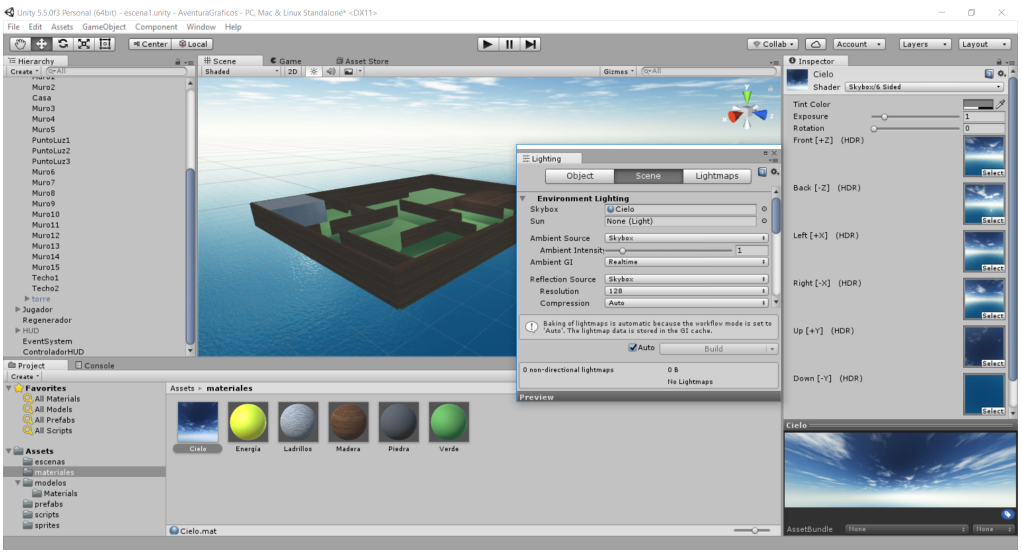


Figura 5.4: Asignación de las texturas del cielo.

5.3. Modelos 3D personalizados

Sustituir los modelos basados en formas geométricas básicas por otros más elaborados puede contribuir en gran medida al acabado general del juego. Para utilizar modelos 3D personalizados hay principalmente dos opciones: obtenerlos de la *Asset Store*, o crearlos con herramientas externas de modelado e importarlos después. La *Asset Store* [22] es un repositorio de recursos

realizados por la comunidad, algunos de los cuales son gratuitos y otros de pago. El catálogo es realmente amplio, aunque si se necesita algo muy específico puede ser necesario recurrir a dichas herramientas para obtener exactamente lo que se busca.

5.3.1. Importación de modelos con Blender

Comenzaremos mostrando como importar modelos 3D personalizados realizados con *Blender* y así remplazar el aspecto que tienen los enemigos y otros elementos. Blender es una herramienta de software libre que permite modelar en 3D, realizar animaciones y todo tipo de tareas relacionadas con la elaboración de gráficos por computador.

En este caso haremos uso de este programa para realizar modelos algo más complejos que los empleados hasta ahora. No se entrará en mucho detalle acerca de cómo realizar estos modelos, ya que se trata de una herramienta ajena al propio Unity, pero el proceso consiste básicamente en componer modelos más elaborados a partir de formas básicas y de operaciones como escalados, traslaciones y rotaciones. También es posible definir los materiales del modelo en Blender, o bien importar el modelo sin materiales y crearlos o modificarlos en Unity posteriormente.

Es importante tener en cuenta a la hora de modelar el objeto que Blender utiliza un sistema de coordenadas distinto (dextrógiro, a diferencia de Unity que usa uno levógiro). Por ello es importante que la orientación del objeto coincida con la esperada en Unity, especialmente si el objeto va a estar en movimiento, ya que modificar los ejes locales del objeto después de haberlo exportado puede ser más complicado.

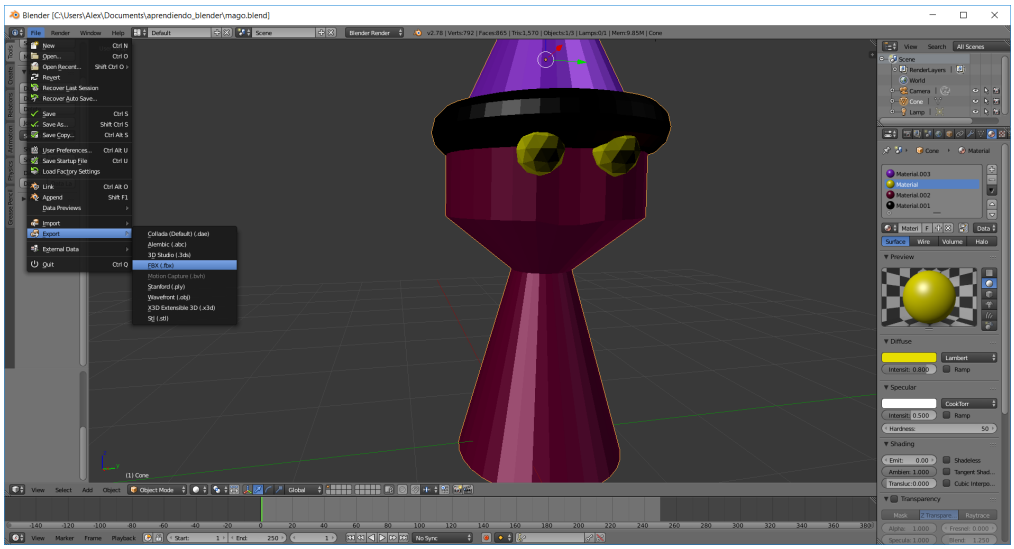


Figura 5.5: Exportación de modelo 3D en Blender.

Una vez tengamos el modelo preparado, por ejemplo el de un enemigo, Blender nos permite exportarlo como archivo FBX, mediante *File* → *Export* → *FBX* (figura 5.5). De vuelta en Unity, crearemos un nuevo directorio, *modelos*, en el cual importaremos este archivo con la opción *Assets* → *Import New Asset*. Junto al modelo importado se creará automáticamente otro directorio, *Materials*, con los materiales utilizados, si es que se ha empleado alguno.

El siguiente paso es sustituir el prefab que hemos estado utilizando para los enemigos por otro que utilice el nuevo modelo. Para ello añadimos el modelo a la escena y le asignamos el script *IAEnemigo.cs*, así como el prefab del proyectil al script para que pueda dispararlo. El nuevo modelo no tendrá asociado ningún *Collider*, que será necesario para la detección de los golpes del jugador, asi que le asociamos uno con forma de cápsula, seleccionando el objeto y seleccionando *Add Component* → *Physics* → *Capsule Collider*. Mediante el botón *Edit Collider* tendremos que modificar el tamaño del *Collider* para que se ajuste aproximadamente al del modelo (figura 5.6). Tras estos pasos podemos guardar el modelo añadido a la escena como prefab, y asignar este mismo prefab al objeto regenerador de enemigos. Este nuevo modelo sustituirá a los enemigos de la escena.

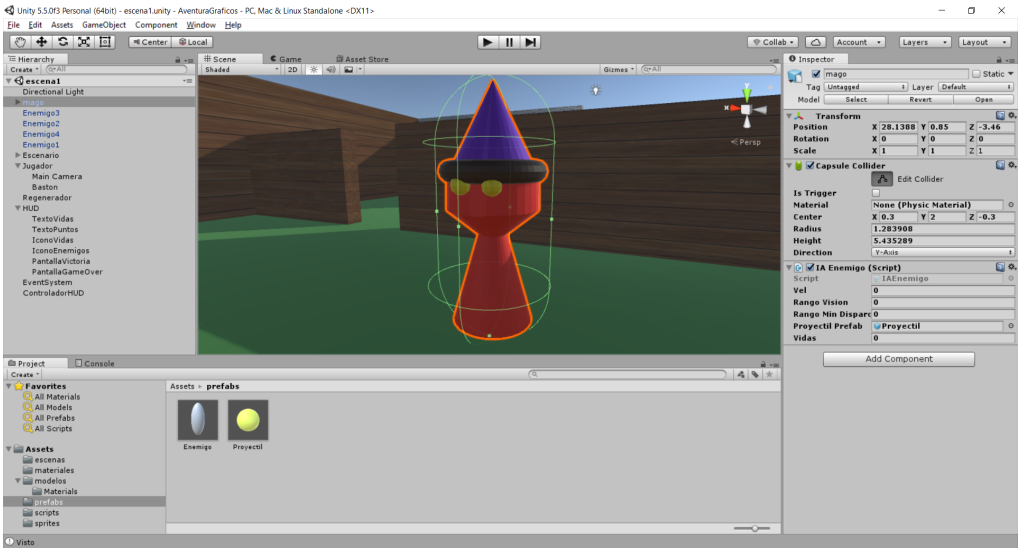


Figura 5.6: Ajuste de Collider al nuevo modelo.

Se seguirá el mismo procedimiento para sustituir los modelos del bastón del jugador y de la torre del escenario, si bien en este caso no hará falta añadir *colliders*. En el caso del bastón, nos aseguraremos que tenga asignado su script correspondiente, *MovimientoBaston.cs*.

5.3.2. Importación de modelos de la Asset Store

Como alternativa a crear los modelos y materiales desde cero, desde la *Asset Store* de Unity se pueden importar todo tipo de componentes realizados por artistas y usuarios de la comunidad. Se puede acceder a ella desde la propia herramienta o desde un navegador web. Para nuestra aplicación importaremos uno de los paquetes de modelos gratuitos que están disponibles para añadir una casa a la escena [23].

Después de encontrar el recurso que se quiera utilizar, basta con pulsar el botón de Importar para que se añadan los elementos seleccionados (figura 5.7).

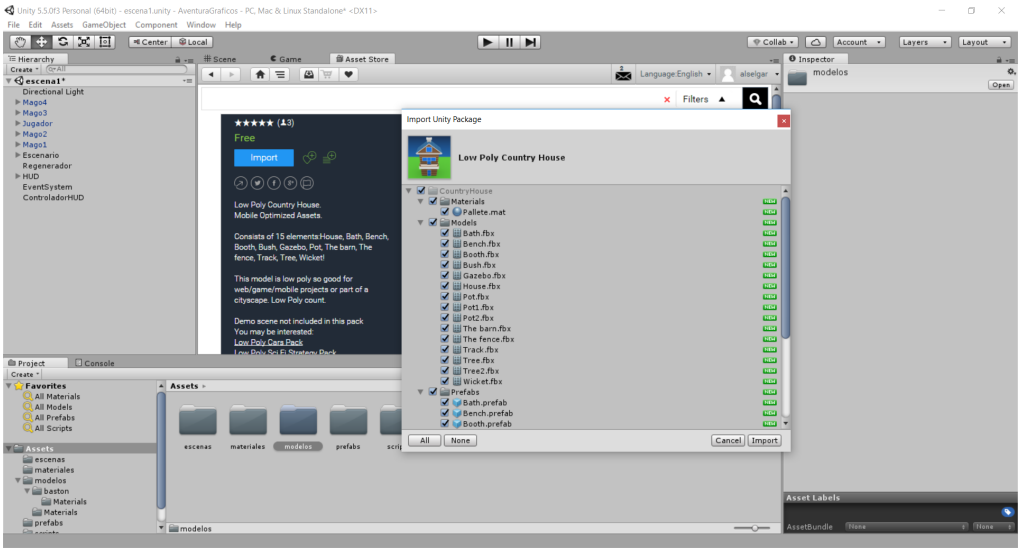


Figura 5.7: Selección de recursos a importar.

Entre los ficheros del proyecto aparecerá un nuevo directorio con los materiales, modelos y prefabs del paquete. Para añadir la casa a la escena, arrastramos su prefab directamente como cualquier otro recurso. El resultado se puede ver en la figura 5.8.

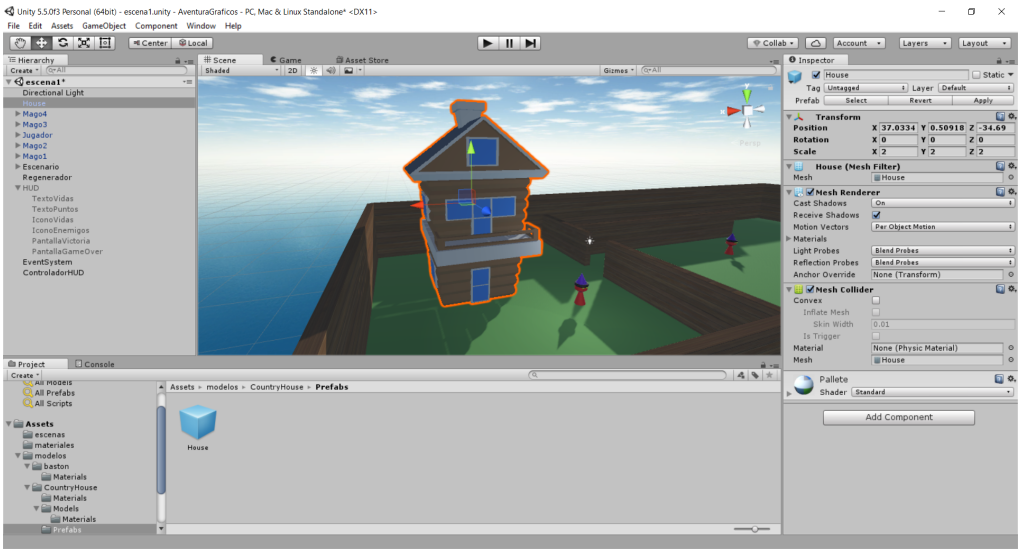


Figura 5.8: Modelo importado desde la Asset Store.

5.4. Efectos de partículas

El siguiente paso será añadir un efecto de partículas a los proyectiles enemigos. Con la opción *GameObject* → *Particle System* podemos añadir un sistema de partículas a la escena. Con los valores por defecto, las partículas son de color blanco y se expanden en forma de cono. Si seleccionamos el objeto, en el inspector tenemos muchos parámetros que podemos ajustar para lograr el aspecto deseado. En este caso cambiaremos los valores *Start Lifetime* y *Start Speed*, que regulan el tiempo de vida de cada partícula y su velocidad inicial; *Start Color*, para darle una tonalidad amarilla (figura 5.9); y *Simulation Space* lo estableceremos a *World* para que el flujo de partículas se mueva junto con el proyectil. En la pestaña *Shape*, cambiaremos la forma de *Cone* a *Cube* y reduciremos su tamaño para estrechar la anchura de emisión. Otros parámetros importantes son *Size over Lifetime*, donde se puede definir con una gráfica como varía el tamaño de cada partícula en el tiempo, y *Material* (dentro de la pestaña *Renderer*) que nos permite seleccionar un material específico para las partículas.

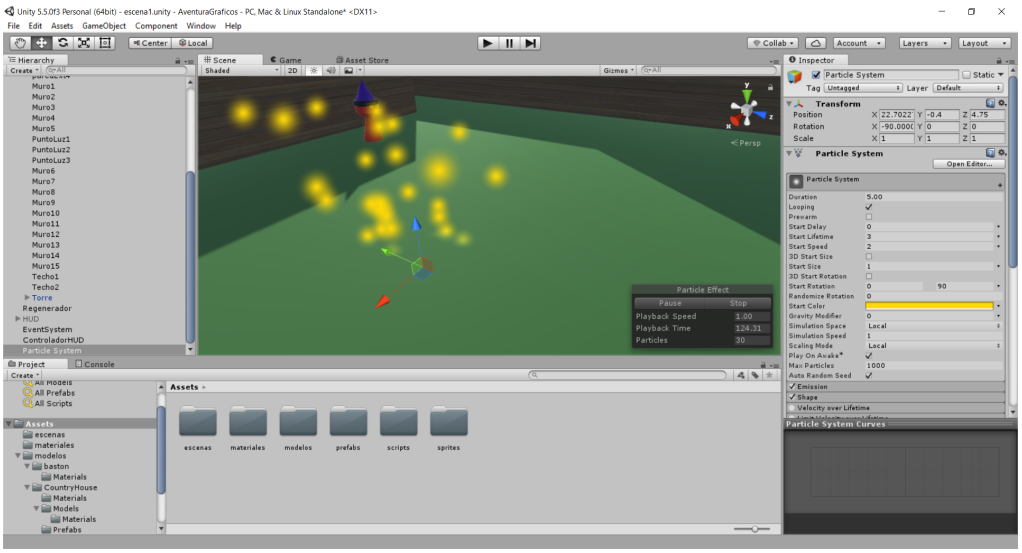


Figura 5.9: Sistema de partículas.

Para este último parámetro crearemos un nuevo material, *Estrellas*, a partir de un sprite, de la misma forma que se hizo en apartados anteriores. El sprite será un dibujo de una de las partículas, como muestra la figura 5.10.

Figura 5.10: Sprite de partícula.

En el inspector de este nuevo material, debemos cambiar el Shader por defecto a *Particles/Aditive(soft)* para que las nuevas partículas se muestren correctamente. El último paso es asignar este material al sistema de partículas en la casilla *Material* del inspector, y hacer que el propio sistema de partículas sea hijo del prefab de proyectil enemigo. El resultado final se puede apreciar en la figura 5.11.

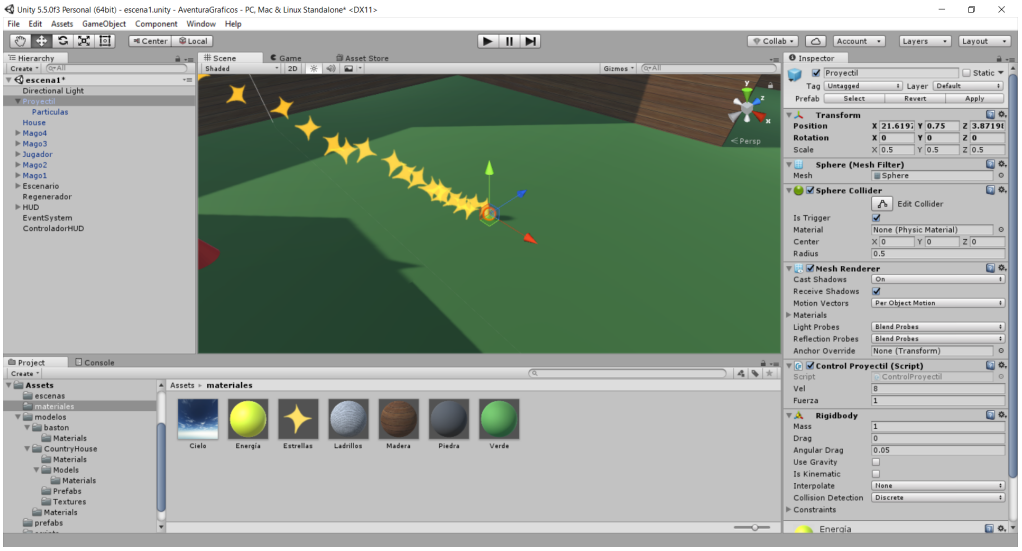


Figura 5.11: Sistema de partículas con material aplicado.

5.5. Sonidos

Para acabar puliendo el acabado del juego añadiremos algunos sonidos al juego. Uno de ellos lo asociaremos también a los proyectiles enemigos. Para ello importaremos un clip de sonido. De la misma forma que con las imágenes, Unity admite muchos formatos distintos, y al realizar la importación Unity aplica una compresión del audio. Crearemos un nuevo directorio para almacenar estos clips de sonido, e importaremos uno que sea adecuado [24] siguiendo el mismo procedimiento que con cualquier otro asset (mediante *Assets* → *Import New Asset* o directamente arrastrando el archivo).

A continuación accederemos al prefab del proyectil y añadiremos un nuevo componente de tipo AudioSource, mediante *Add Component* → *Audio* → *Audio Source*. Con esto estamos especificando que el proyectil es una fuente de sonido. En las propiedades de este componente, podemos asignar el sonido importado asignándolo a la casilla *AudioClip* (figura 5.12). También existen otros parámetros para controlar el volumen o reproducir en bucle, por ejemplo. En este caso dejaremos el resto de parámetros a su valor por defecto. Con estos cambios cada proyectil tendrá asociado su sonido correspondiente.

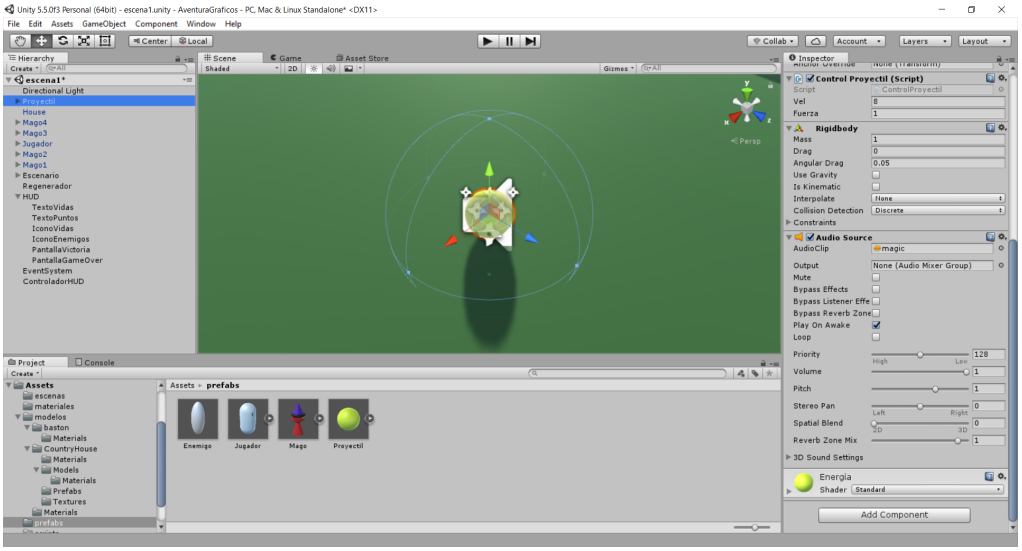


Figura 5.12: Configuración de AudioSource del proyectil.

Seguiremos el mismo proceso para asignar un sonido a las pantallas de victoria y de derrota, accediendo a estos objetos en el HUD y añadiéndoles el componente *AudioSource* de la misma manera [25] [26].

También añadiremos un componente *AudioSource* al objeto del jugador para reproducir una pista de música [28]. Al importar este sonido, marcaremos la casilla *Load In Background*, que permite que se cargue en segundo plano para mejorar el rendimiento. Hacer esto suele ser recomendable para clips de sonido de cierta duración, como la música. En las propiedades del *AudioSource*, habilitaremos *Play On Awake*, que hace que comience la reproducción al arrancar el juego, y *Loop*, para que se reproduzca repetidamente.

5.5.1. Reproducción de sonidos por eventos

Si se pretende reproducir un sonido respondiendo a algún evento dentro del juego, es necesario hacerlo desde código. Para ello, modificaremos el script *IAEnemigo.cs* para que los enemigos emitan un sonido al recibir un golpe [27]:

IAEnemigo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IAEnemigo : MonoBehaviour {
   [...]
   [SerializeField] private AudioSource fuenteSonido;
   [SerializeField] private AudioClip sonidoGolpe;

   [...]

   public void RecibirGolpe() {
      vidas--;
      if (vidas < 1) {
         fuenteSonido.PlayOneShot(sonidoGolpe);
         StartCoroutine(Desaparecer());
      [...]
      }
   }
```

Al principio del script se declaran atributos para recibir un objeto que sea la fuente del sonido y otro para el propio clip de sonido. Cuando el enemigo recibe un golpe, se llama al método *PlayOneShot()* para reproducirlo.

De vuelta en Unity, añadiremos un componente AudioSource al prefab del enemigo y dejaremos todos los valores por defecto. En el inspector del script *IAEnemigo.cs*, asignaremos como AudioSource el prefab del enemigo (para poder hacerlo es necesario que tenga el componente AudioSource que le hemos añadido) y como AudioClip el clip de sonido que queramos reproducir.

Con todos los cambios realizados en el aspecto gráfico y sonoro, el juego tiene un acabado mejor y los elementos de la escena se distinguen con mayor facilidad que cuando estaban representados por formas básicas con texturas en blanco.

5.6. Menú de opciones

Con las funcionalidades principales añadidas, además del sonido y la mejora de los gráficos, se ha optado por añadir un menú para poder configurar las siguientes opciones durante una partida:

- Mostrar u ocultar la tasa de frames por segundo (FPS).
- Activar y desactivar el sonido.
- Mostrar los créditos.
- Salir de la aplicación.

El menú podrá abrirse en cualquier momento pulsando la tecla *ESC* del teclado y pausará la partida hasta que se cierre. Para implementarlo, en primer lugar añadiremos un elemento de tipo *Text* al HUD que mostrará la tasa de frames por segundo. Además, añadiremos también un *GameObject* vacío, que llamaremos *Menu*, y que tendrá como hijos a los botones que activarán las distintas opciones. Estos botones son elementos que incorporaremos al menú mediante *GameObject* → *UI* → *Button*.

La programación de las opciones disponibles se hará a través del script *ControlHUD.cs*, al cual añadiremos métodos que responderán a las pulsaciones de los botones. Los métodos añadidos se muestran a continuación:

ControlHUD.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class ControlHUD : MonoBehaviour {

   [SerializeField] private Text textoVidas;
   [SerializeField] private Text textoPuntos;

   [SerializeField] private Text contadorFPS;
   private float intervaloActFPS = 0.5f;
   private float tiempoAcum = 0.0f;
```

```
private float fps = 0.0f;
private bool mostrarFPS = false;

[SerializeField] private GameObject menu;
[SerializeField] private GameObject jugador;
[SerializeField] private GameObject baston;
[SerializeField] private GameObject camara;
[SerializeField] private GameObject creditos;
private bool sonidoActivado = true;

[...]

void Start() {
   [...]
   menu.SetActive(false);
   creditos.SetActive(false);
   Cursor.visible = false;
}

void Update() {

   // ESC abre el menu
   if (Input.GetKey(KeyCode.Escape)) {
      abrirMenu();
   }

   // calcula FPS
   if (mostrarFPS) {
      tiempoAcum += Time.deltaTime;

      if (tiempoAcum > intervaloActFPS) {
         fps = 1.0f / Time.deltaTime;
         tiempoAcum = 0.0f;
      }
      contadorFPS.text = "FPS: " + fps.ToString("F2");
   }
   else {
      contadorFPS.text = "";
   }

}

public void botonSalir() {
```

```
    Application.Quit();
}

public void botonFPS() {
    mostrarFPS = !mostrarFPS;
}

public void abrirMenu() {
    Time.timeScale = 0.0f;
    Cursor.visible = true;
    jugador.GetComponent<MirarAlrededor>().enabled = false;
    baston.GetComponent<MovimientoBaston>().enabled = false;
    camara.GetComponent<MirarAlrededor>().enabled = false;
    menu.SetActive(true);
    creditos.SetActive(false);
}

public void cerrarMenu() {
    Time.timeScale = 1.0f;
    Cursor.visible = false;
    jugador.GetComponent<MirarAlrededor>().enabled = true;
    baston.GetComponent<MovimientoBaston>().enabled = true;
    camara.GetComponent<MirarAlrededor>().enabled = true;
    menu.SetActive(false);
}

public void mostrarCreditos() {
    menu.SetActive(false);
    creditos.SetActive(true);
}

public void cerrarCreditos() {
    creditos.SetActive(false);
    menu.SetActive(true);
}

public void botonSonido() {
    sonidoActivado = !sonidoActivado;
    if (sonidoActivado) {
        AudioListener.volume = 1;
    }
    else {
        AudioListener.volume = 0;
```

```
        }
    }
    [...]
}
```

Como se puede observar en el código, la mayoría de estas funciones habilitan o deshabilitan los componentes del menú que se quieren mostrar en cada momento. Dado que tenemos todos los botones agrupados como objetos hijos de *Menu*, cambiar la visibilidad de este objeto hace lo mismo con el menú completo. En cuanto a *Update()*, aquí se ha añadido la lógica que calcula la tasa de frames por segundo. Para evitar que el indicador cambie demasiado rápido, se actualizará cada medio segundo.

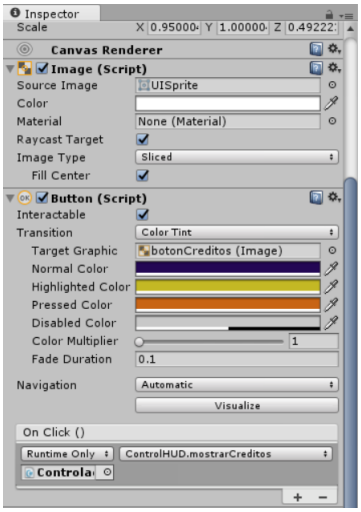


Figura 5.13: Selección de método asociado a un botón.

Como último paso, tendremos que asociar cada botón con el método correspondiente. Para ello, seleccionaremos cualquier botón para acceder al inspector (figura 5.13). Aquí podemos modificar parámetros como el aspecto del botón o si se quiere utilizar una imagen en su lugar, por ejemplo. En la parte inferior, en la propiedad *OnClick()*, tenemos una casilla donde tendremos que seleccionar el objeto que tenga asociado el script *ControlHUD.cs*, que en este caso es *ControladorHUD*. Tras hacerlo, podremos seleccionar en el desplegable de la derecha cualquier método del script, que se ejecutará al pulsar del botón. El resultado se muestra en la figura 5.14.

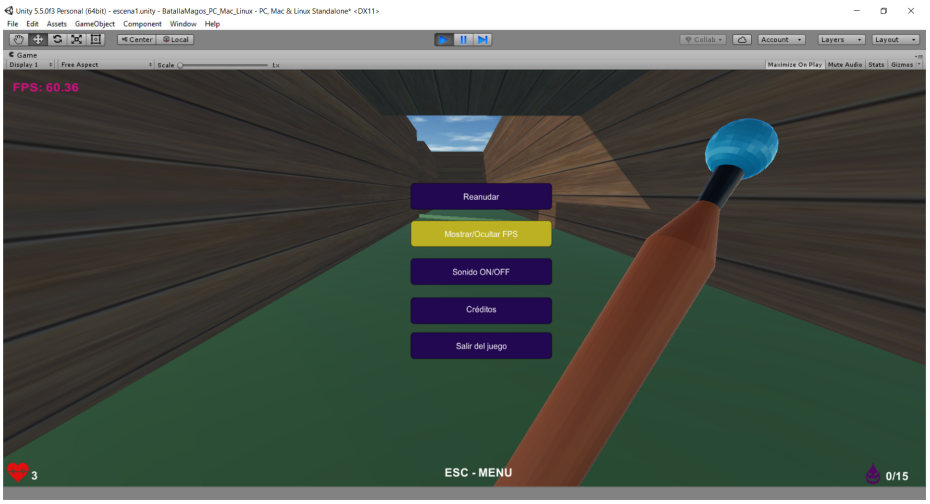


Figura 5.14: Menú de opciones.

Capítulo 6

Evolución de la Realidad Virtual y opciones disponibles en Unity

6.1. Definición e historia

La realidad virtual consiste en un conjunto de tecnologías, generalmente relacionadas con la computación, que simulan un entorno real o imaginario a través de un casco o visor que genera imágenes, sonidos u otras sensaciones. El usuario de estas tecnologías es capaz de mirar alrededor en un entorno inmersivo generado por ordenador, y a menudo interactuar con sus elementos.

No hay un acuerdo acerca del origen exacto de la realidad virtual, pero los primeros planteamientos a nivel teórico se remontan a la década de los años 50 [29]. En 1962, Morton Heilig construyó un dispositivo mecánico al que llamó *Sensorama*, que podía reproducir cinco películas de corta duración y estimular los sentidos de la vista, el oído, el olfato y el tacto (figura 6.1).

Durante esa misma época, Douglas Engelbart utilizó pantallas de computadores como dispositivos de entrada y salida. En 1968, Ivan Sutherland diseñó con ayuda de su alumno Bob Sproull el primer casco de realidad virtual y realidad aumentada. Era bastante primitivo en cuanto a realismo, ya que los gráficos generados estaban compuestos únicamente por figuras geométricas en *wireframe*, y era tan pesado que tenía que estar suspendido del techo. Por esta razón, este dispositivo recibió el nombre de *La espada de Damocles*.

En las décadas de 1970 y 1980, destaca el *Aspen Movie Map* creado por el

Figura 6.1: Sensorama. *Fuente: wikipedia.org.*

MIT (Instituto Tecnológico de Massachusetts), que fue una recreación virtual en la que los usuarios podían recorrer las calles de Aspen, Colorado. Las dos primeras recreaciones estaban hechas con fotografías, y la tercera con gráficos generados por computador. La compañía Atari fundó un laboratorio de investigación de realidad virtual en 1982, que tuvo que cerrar tras dos años de actividad debido a la crisis de la industria de los videojuegos de 1983 [29]. Sin embargo, sus empleados continuaron con sus investigaciones en tecnologías relacionadas con la realidad virtual.

En el año 1991, se creó la primera habitación cúbica inmersiva, conocida como *The Cave* [30]. Consistía en un entorno con múltiples imágenes proyectadas que permitía que el usuario viera su propio cuerpo en relación al de otros dentro de la habitación.

En ese mismo año, Sega anunció el Sega VR Headset para maquinas recreativas [31] y su consola Mega Drive. Tenía pantallas LCD, sonido estéreo y sensores que detectaban la posición de la cabeza del usuario. La compañía Virtuality [32] lanzó el primer sistema VR multijugador, que incluía un casco y guantes para múltiples usuarios. El sistema completo tenía un coste de $73.0000.

En el año 1995 Nintendo lanzó su consola *Virtual Boy* (figura 6.2), que si bien no se considera una máquina de realidad virtual, era capaz de mostrar 3D estereoscópico mediante un efecto de paralaje. Este año también se lanzó el casco de realidad virtual VFX-1 (figura 6.3), que funcionaba conectado a un PC y era compatible con juegos como *Descent*, *System Shock* o *Quake*.

Figura 6.2: Consola Virtual Boy. *Fuente: pixfans.com.*

Ya en la década de los 2000, en 2001 se diseña la primera habitación inmersiva que funciona con un PC, la SAS3 o SAS Cube. En 2010, Palmer Luckey diseña el primer prototipo de *Oculus Rift*, que como principal novedad ofrecía un campo de visón de 90º. En 2013, Valve hizo importantes progresos en la eliminación del retardo en la visualización de contenido en realidad virtual, y en 2014 mostró un prototipo de *SteamSight*, un dispositivo con pantallas separadas para cada ojo de resolución 1K y lentes de Fresnel, que se caracterizan por tener una gran apertura y corta distancia focal.

En 2014, Facebook compró *Oculus VR* y Sony anunció *Project Morpheus*, que más tarde renombraría como PlayStation VR, un casco de realidad virtual para la consola PlayStation 4. Este año Google también anuncia *Cardboard*, un visor que el propio usuario puede construirse con una plantilla de cartón, y que sirve de soporte a un teléfono móvil con el que se visualiza el contenido en VR.

En 2015, HTC y Valve anuncian *HTC Vive*, un conjunto de dispositivos que incluyen un casco de realidad virtual, controladores, y sensores que pueden repartirse por una habitación para registrar la posición del usuario

mediante infrarojos. En 2016 se repartieron las primeras unidades de este dispositivo para usarse con la plataforma Steam VR.

Figura 6.3: Casco VFX-1. *Fuente: wikipedia.org.*

En la actualidad, hay una gran cantidad de compañías desarrollando productos relacionados con la realidad virtual, incluyendo grandes empresas con departamentos dedicados exclusivamente a este área, como Google, Apple, Amazon, Microsoft, Sony o Samsung. Hay ciertos aspectos en los que se pueden hacer grandes avances, como los controladores hápticos, que no están muy desarrollados todavía, la resolución de la imagen o la tasa de refresco. La realidad virtual hoy en día tiene aplicación en muchos campos: en la industria del entretenimiento (cine y videojuegos principalmente), medicina (tratamiento de enfermedades), en la educación y entrenamiento para todo tipo de actividades, en ingeniería y arquitectura, arte, marketing, etc.

6.2. Plataformas actuales de realidad virtual y soporte en Unity

Unity permite desarrollar aplicaciones para las principales plataformas actuales de realidad virtual y realidad aumentada. A continuación se describirán las características de estas plataformas.

6.2.1. Oculus Rift

Oculus Rift (figura 6.4) [34] es un casco de realidad virtual desarrollado por Oculus VR, una compañía independiente que fundó este proyecto con

éxito en 2012 a través de la plataforma *Kickstarter*. La compañía fue comprada posteriormente por Facebook en Marzo de 2014.

Figura 6.4: Oculus Rift. Fuente: wikipedia.org

Antes de ser lanzada al público, se distribuyeron varios prototipos como kit para desarrolladores para que hubiese aplicaciones listas para el lanzamiento. Desde entonces, posteriores revisiones han mejorado sus características técnicas.

El modelo actual, conocido como *Consumer Version 1*, tiene las siguientes características:

- Pantalla OLED estereoscópica con una resolución total de 2160x1200 (1080x1200 por cada ojo).
- 90 Hz de tasa de refresco.
- Campo de visión de 110º.
- Separación entre lentes ajustable.
- Auriculares de audio 3D integrados.
- Admite el uso de controladores (figura 6.5).

El sistema de posicionamiento de Oculus Rift recibe el nombre de *Constellation* y funciona mediante un dispositivo detector de infrarojos que se conecta al casco mediante una conexión USB. Este dispositivo detecta la luz que emiten unos LED de infrarojos integrados en el casco. Este casco detecta la posición y rotación de la cabeza mediante 6 ejes. En cuanto a los controladores, son dos dispositivos con botones y sensores para ser rastreados de la misma forma que el casco. También se admite el uso del controlador de

Xbox One, que viene incluido con cada unidad de Oculus Rift. El coste del casco con los periféricos necesarios es de 590 euros aproximadamente, y el de los mandos (opcionales) es de 120 euros.

Figura 6.5: Controladores de Oculus Rift.

Como requisitos para utilizar esta plataforma, la compañía recomienda un PC con procesador Intel Core i3-6190 o AMD FX 4350, tarjeta gráfica Nvidia GeForce GTX 960 o equivalente, 3 puertos USB y sistema operativo Windows 8 o superior.

6.2.2. Google Cardboard

Google Cardboard es una plataforma desarrollada por Google y destinada a emplear como visor cualquier teléfono compatible junto con un soporte de cartón que incluye unas lentes y un gatillo magnético (figura 6.6). Su principal ventaja es que es una forma accesible y económica de obtener un dispositivo de realidad virtual, ya que cualquier persona puede realizar el montaje del visor a partir de los materiales necesarios.

Tras montar el soporte de cartón con las lentes, el teléfono se introduce en la parte posterior. Las aplicaciones desarrolladas para esta plataforma dividen la pantalla en dos y aplican un efecto de distorsión que, junto con el efecto de las lentes, produce una imagen estereoscópica con un campo de visión bastante amplio.

Existen muchos modelos de visor disponibles, y cada uno determina los requisitos necesarios del teléfono que se vaya a utilizar, si bien en la mayoría

de los casos es necesario que el terminal tenga giroscopio y sensor magnético para poder detectar las pulsaciones del gatillo. En cuanto al sistema operativo del teléfono, se requiere Android 4.1 o bien iOS 8.0 (o versiones posteriores). Existen algunas aplicaciones que pueden ser útiles para averiguar si un teléfono en concreto es compatible, como por ejemplo *EZE VR*, disponible en la *Play Store*.

Figura 6.6: Visor Cardboard. *Fuente: wikipedia.org.*

6.2.3. Daydream

Daydream es una plataforma lanzada por Google en Noviembre de 2016 que también hace uso de dispositivos móviles, aunque requiere de un visor específico que viene acompañado de un controlador (figura 6.7). El objetivo de Daydream es ofrecer una experiencia de mejor calidad que en Cardboard, ya que el controlador ofrece más posibilidades de interacción que el gatillo magnético de este y el visor está hecho con materiales más duraderos, motivo por el cual no es una opción tan económica como Cardboard. Tanto el teléfono como el visor deben de cumplir con una serie de requisitos [33], si bien Google vende su propio visor oficial por un precio aproximado de $80 (figura 6.7).

El controlador incluido con el visor es inalámbrico y tiene un panel táctil, un botón para acceder al menú, otro para funciones específicas de cada aplicación y dos botones para regular el volumen. Además incluye sensores que son capaces de detectar la orientación del controlador y la posición de la mano del usuario.

Figura 6.7: Visor Daydream con controlador. *Fuente: elpais.com.*

Daydream requiere Android 7.1 (Nougat) o superior, y de hecho viene integrado con esta versión del sistema operativo.

6.2.4. Steam VR/HTC Vive

HTC Vive (figura 6.8) es un casco de realidad virtual desarrollado por Valve Corporation y HTC, que sirve como periférico para acceder a la plataforma Steam VR. Se puso a la venta en Abril de 2016.

Figura 6.8: Dispositivos de HTC Vive. *Fuente: computerworld.dk.*

Sus características técnicas son:

- Dos pantallas OLED individuales para cada ojo con resolución de 1080x1200 (resolución total de 2160x1200).

- 90 Hz de tasa de refresco.
- Campo de visión de 110º.
- Más de 70 sensores incluyendo giroscopio, acelerómetro y sensores de posición por láser.

El casco viene acompañado de dos controladores y dos bases encargadas de rastrear la posición del jugador dentro de la habitación mediante barridos por lásers infrarrojos. Los objetos capturados tienen fotosensores que son detectados por estos barridos. El área rastreada por el sistema tiene una superficie de 4.6 x 4.6 metros aproximadamente. El propio casco dispone de una cámara frontal para detectar otros objetos en la habitación y alertar si el usuario se acerca al límite de esta superficie (sistema de seguridad *Caperone*) [36].

El sistema funciona conectado a la plataforma Steam VR ejecutándose en un sistema Windows. Las especificaciones recomendadas por el fabricante son un procesador Intel Core i5-4590 o AMD FX 8350, tarjeta gráfica Nvidia GeForce GTX 1060 o AMD Radeon RX 480 y 4GB de RAM (o especificaciones superiores). También se requiere una salida HDMI 1.4 o DisplayPort 1.2, un puerto USB 2.0 y sistema operativo Windows 7 (SP 1), Windows 8.1 o Windows 10.

El coste del set con todos los accesorios necesarios, incluyendo el casco, los dos controladores y las dos bases, es de unos $800.

6.2.5. PlayStation VR

Sony lanzó el casco de realidad virtual PlayStation VR (figura 6.9), conocido durante su desarrollo como Project Morpheus, en Octubre de 2016. El casco está diseñado para conectarse directamente a una consola PlayStation 4 y funciona junto con el controlador DualShock 4 o PlayStation Move.

Sus características técnicas son:

- Pantalla OLED de 5.7 pulgadas con una resolución de 1920x1080 píxeles (960x1080 por cada ojo).
- Tasa de refresco de 120 Hz.
- Campo de visión de 100º.

Figura 6.9: Casco de PlayStation VR. *Fuente: playstation.com.*

- Sistema de rastreo de seis ejes (6DOF).

El sistema de rastreo funciona con 9 LEDs situados en el casco que son detectados por una PlayStation Camera. Las imágenes mostradas por el casco pueden mostrarse simultáneamente en un televisor, con aplicaciones en juegos cooperativos o competitivos.

Los desarrolladores para esta plataforma pueden elegir entre tres modos de renderizado: nativo a 60 Hz, nativo a 120 Hz y un modo que realiza una conversión de juegos que funcionen a 60Hz a los 120 Hz, mediante una técnica de interpolación [37].

Los requisitos para utilizar esta plataforma son una consola PlayStation 4, el casco de realidad virtual y una Playstation Camera, siendo opcionales los mandos Dualshock 4 y PlayStation Move, según cada juego. El casco tiene un precio de 400 euros aproximadamente.

6.2.6. Samsung Gear VR

Samsung Gear VR (figura 6.10) es un visor de realidad virtual que, de forma similar a las plataformas Cardboard o Daydream, utiliza un teléfono Samsung compatible como dispositivo de visualización. El casco se conecta al teléfono mediante un puerto micro USB, y además tiene sus propios sensores de detección de rotación, un panel táctil, y un botón en un lateral. El área de visión en el modelo más reciente es de 101°. El casco viene acompañado de un controlador con su propio panel táctil, un gatillo y botones de navegación

y volumen [38].

Figura 6.10: Samsung Gear VR. *Fuente: samsung.com.*

Los teléfonos compatibles son los siguientes:

- Samsung Galaxy S6 / S6 Edge / S6 Edge+.
- Samsung Galaxy Note 5.
- Samsung Galaxy S7 / S7 Edge.
- Samsung Galaxy S8 / S8+.

Este dispositivo tiene un precio aproximado de 100 euros.

6.2.7. Microsoft HoloLens

Microsoft Hololens (figura 6.11) son una gafas inteligentes (*smartglasses*) de realidad mixta desarrolladas por Microsoft, todavía en fase de desarrollo.

Figura 6.11: Microsoft Hololens. *Fuente: microsoft.com.*

Si bien la realidad aumentada permite ver elementos virtuales generados por ordenador en un entorno real, la realidad mixta genera entornos a partir de elementos tanto del mundo real como del virtual, y permite que estos elementos interactúen entre si.

Estas gafas se ajustan a la cabeza del usuario y disponen de múltiples sensores en la parte frontal, donde además se sitúan varias cámaras y procesadores. En el visor, que está tintado, hay unas lentes donde se proyectan las imágenes generadas por el dispositivo. En la parte lateral hay unos altavoces de audio 3D diseñados para que los sonidos puedan diferenciarse con facilidad de los del mundo real, ademas de varios botones para ajustar el volumen o el brillo.

Sus principales características técnicas son [39]:

- Sensores frontales: acelerómetro, magnetómetro, giroscopio y cuatro sensores de reconocimiento del entorno.
- Cámara de profundidad con resolución holográfica de 2.3M.
- Cámara fotográfica de 2.4 M.
- Micrófonos y reconocimiento de voz.
- Sensor de luz ambiental.
- Procesador Intel 32 bit a 1Ghz.
- Coprocesador HPU (Holografic Processing Unit).
- 2 GB de RAM.
- 64 GB de almacenamiento en memoria flash.
- Conectividad por WiFi y Bluetooth.
- Batería recargable por puerto micro USB.

En Marzo de 2016 se empezaron a distribuir unidades para desarrolladores a un precio de $3000, y la fecha en que se lanzará una versión para usuarios finales todavía está por determinar.

6.3. Soporte en Unity para plataformas de realidad virtual

Como se ha comentado anteriormente, Unity permite desarrollar aplicaciones de realidad virtual para las principales plataformas actuales. La compañía destaca la capacidad del motor para conseguir un renderizado optimizado y altas tasas de refresco, versatilidad entre las distintas plataformas mediante el uso de herramientas de desarrollo específicas (SDKs) y una estrecha colaboración con los fabricantes de los distintos dispositivos.

Hasta hace poco Unity daba soporte a cada plataforma únicamente mediante la importación de plugins o SDKs específicos, pero en futuras versiones el motor favorecerá el uso del soporte nativo incorporado, que no requiere la importación de estos paquetes y ofrece funcionalidades básicas de realidad virtual [40]. Esto se consigue habilitando la opción *Virtual Reality Supported* en la configuración de *Player Settings* (figura 6.12) y a continuación eligiendo la plataforma deseada en el menú desplegable. Este método sería el mismo para cualquier plataforma con soporte nativo disponible. Sin embargo, para el uso de ciertas funciones, y dependiendo de la plataforma, todavía puede ser necesaria la importación del SDK correspondiente, por ejemplo si se va a dar soporte a controladores de Daydream o se quieren utilizar ciertos prefabs incluidos en el kit de desarrollo [44]. En este caso la opción *Virtual Reality Supported* deberá estar desactivada.

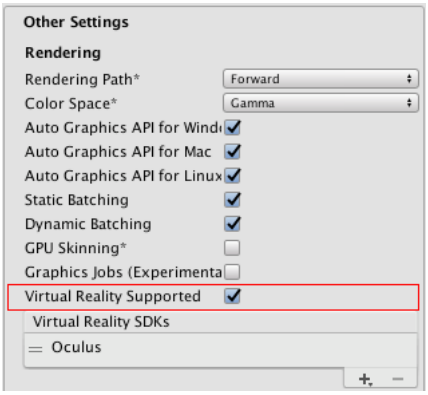


Figura 6.12: Opción *Virtual Reality Supported* en Player Settings.

En el siguiente capítulo se describirá cómo llevar a cabo una conversión de la aplicación desarrollada a la realidad virtual en la plataforma Cardboard mediante el SDK Google VR para Unity, abordando dos adaptaciones

a dispositivos con métodos de control distintos. La importación del SDK será necesaria en este caso, porque la versión de Unity utilizada en este proyecto, la 5.5, todavía no dispone de soporte nativo que funcione correctamente para esta plataforma.

El SDK de Google VR está disponible para Unity, Android, iOS y Unreal Engine 4, y da soporte tanto para Cardboard como para Daydream en todas estas plataformas excepto en iOS, donde sólo se da soporte a Cardboard [41].

Importar este kit de desarrollo nos permitirá usar los componentes necesarios para que la aplicación haga uso de realidad virtual, como prefabs y scripts que incluyen funcionalidades de visión estereoscópica, eventos, audio para entornos inmersivos, etc. El proceso de importación y uso del SDK también se detallará en el siguiente capítulo.

En futuras versiones de Unity, si se utilizara el soporte nativo, los pasos a seguir serían los mismos o muy similares, pero ya no existiría la necesidad de importar manualmente ningún paquete adicional.

Capítulo 7

Adaptación de la aplicación a dispositivos de Realidad Virtual

En este capítulo se abordará el proceso de adaptación a la realidad virtual de la aplicación 3D desarrollada en capítulos anteriores, usando el SDK de Google VR para Unity y teniendo como objetivo que funcione tanto en visores Cardboard como en visores con controlador.

7.1. Importación del SDK Google VR

En primer lugar, será necesario tener instalados en el sistema el SDK de Android y el JDK (Java Development Kit) de Java, para poder realizar la compilación del ejecutable para Android. Ambos pueden obtenerse fácilmente y de forma gratuita en sus respectivos sitios web [42] [43].

Tras la instalación de ambas herramientas de desarrollo, hay que establecer en Unity la ruta a las librerías de cada una. Esto se puede configurar en el menú *Edit* → *Preferences* → *External Tools*, en los campos *SDK* y *JDK* respectivamente (figura 7.1).

El siguiente paso será importar los paquetes requeridos del SDK de Google VR para Unity según la plataforma. Es importante señalar que este paso es necesario en la versión de Unity utilizada en este proyecto (versión 5.5), pero a partir de la versión 5.6, actualmente disponible en pruebas, Google VR viene integrado de forma nativa y no sería necesario realizar la importación de los paquetes.

Tras descargar el SDK de su sitio web [44], volvemos a Unity y selec-

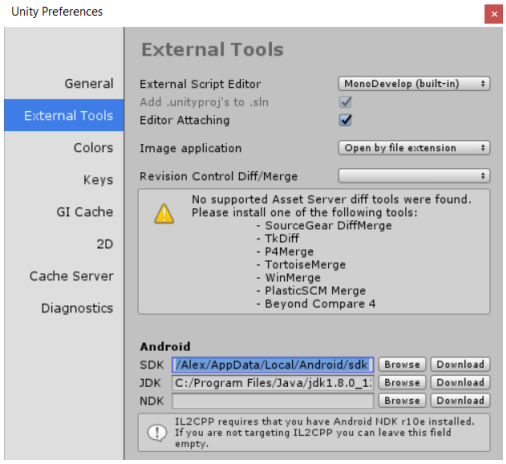


Figura 7.1: Configuración de SDK de Android y JDK.

cionamos *Assets* → *Import New Assets* → *Custom Packages* para indicar la ruta del SDK. Nos aparecerá una ventana desde la que podemos seleccionar individualmente los componentes que deseamos importar al proyecto (figura 7.2). Estos componentes aparecen separados por categorías, y seleccionaremos unas u otras según nuestras necesidades. La categoría *Demos*, por ejemplo, incluye demostraciones para probar algunos componentes de RV, aunque para nuestro proyecto no será necesario importarlos. *Legacy* contiene componentes de versiones anteriores del SDK para Cardboard que están siendo sustituidos, aunque todavía se pueden utilizar en la versión actual. Aunque no utilizaremos ninguno de estos componentes, no importarlos ha ocasionado algunos problemas al ejecutar la aplicación, por tanto los incluiremos. Al hacerlo puede aparecer una pantalla advirtiendo de que se están usando paquetes obsoletos, a pesar de ello continuaremos con el proceso. La categoría iOS tendremos que incluirla si pretendemos distribuir la aplicación en esta plataforma. Tras terminar con la selección pulsamos el botón *Import*.

Cuando haya finalizado el proceso de importación, entre los archivos del proyecto tendremos un nuevo directorio *Google VR*, que contendrá todos los elementos importados.

7.2. Adaptación al visor Cardboard

A la hora de adaptar la aplicación a visores de tipo Cardboard, hay que tener en cuenta que la capacidad que tiene el jugador para interactuar con el

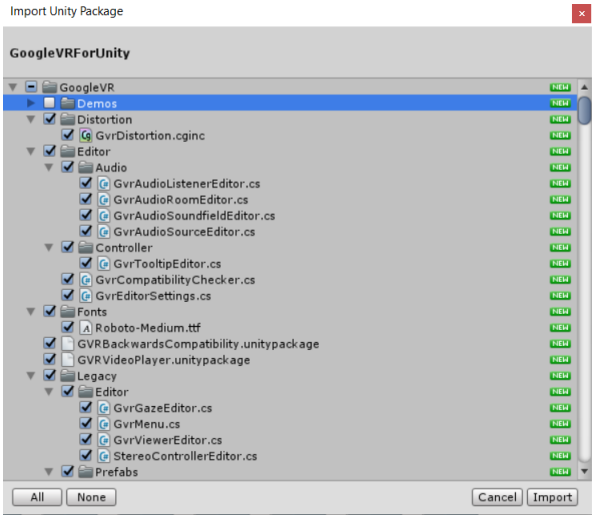


Figura 7.2: Selección de paquetes del SDK Google VR.

entorno se reduce, ya que el visor sólo dispone de un gatillo con dos estados posibles (pulsado o no). Por tanto, además de dotar al juego de visión estereoscópica con el SDK de Google VR, habrá que hacer cambios en algunas de las mecánicas del juego para que sea realmente funcional.

7.2.1. Cambios en las mecánicas

Al no tener disponibles botones de dirección que si podemos aprovechar en un teclado o en cualquier controlador, el principal cambio que se llevará a cabo es que el movimiento del jugador siga un recorrido predeterminado por el escenario. El jugador podrá mirar hacia donde quiera con el movimiento de la cabeza, que será completamente independiente del movimiento de desplazamiento. Puesto que se pierde esta libertad de movimiento, también se pierde la capacidad de acercarse a los enemigos para golpearles desde corta distancia. Por tanto, la forma de derrotar a los enemigos será disparando proyectiles de la misma forma que hacen ellos, usando el gatillo del visor Cardboard.

Como alternativa a esta solución, se ha considerado la posibilidad de hacer que el jugador se mueva en todo momento hacia donde esté mirando. Esta opción tiene como principal ventaja que se conserva el desplazamiento libre, aunque tiene otros inconvenientes. Uno de ellos es que requiere que el jugador que tenga puesto el visor esté de pie y girando sobre si mismo con-

tinuamente para cambiar de dirección. Además el movimiento no se percibe tan natural cuando se está mirando hacia arriba o hacia abajo, ya que en esos casos el desplazamiento sigue siendo horizontal y no coincide con la dirección de la mirada. Por estos motivos finalmente se ha optado por implementar un recorrido predeterminado.

Para llevarlo a cabo, partiremos del proyecto de Unity donde lo dejamos, tras la mejora de los gráficos y la incorporación de sonidos y otros efectos. La implementación del recorrido que realizará el jugador se puede afrontar de varias maneras, en esta ocasión lo haremos mediante puntos de control.

Crearemos estos puntos de control como objetos vacíos (desde el menú, *GameObject* → *Create Empty*) a los cuales añadiremos un componente BoxCollider (desde el inspector del objeto, *Add Component* → *Physics* → *Box Collider*), como muestra la figura 7.3.

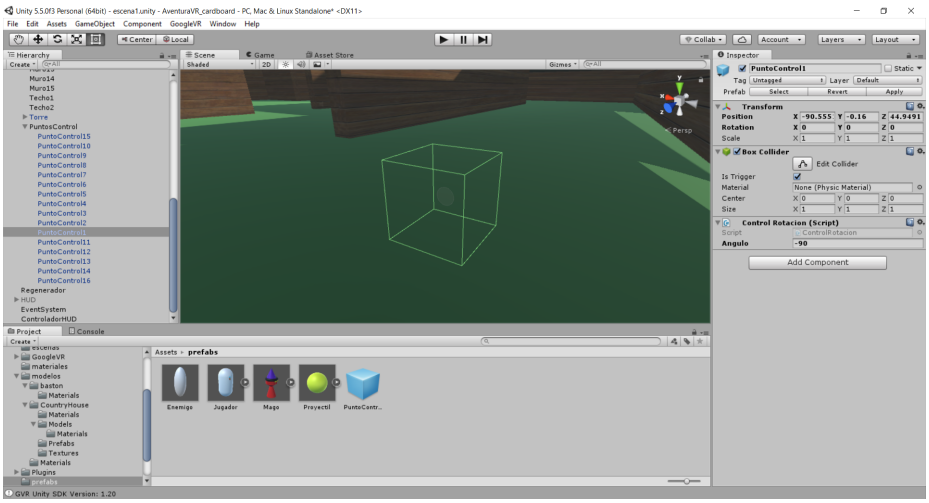


Figura 7.3: Punto de control con BoxCollider.

Estos objetos serán invisibles para el jugador, ya que no tienen una forma ni textura asignada, y los repartiremos por el escenario en puntos determinados para que este efectúe una rotación cuando entre en contacto con ellos. Esto debe ser así porque modificaremos el movimiento del propio jugador para que se mueva de forma continua hacia adelante, por tanto serán los puntos de control los que indiquen al jugador cómo tiene que girar. En las propiedades del BoxCollider, nos aseguraremos que la casilla *IsTrigger* esté activada para que podamos detectar la colisión mediante un script.

Llamaremos a este nuevo script *ControlRotacion.cs* y se lo asignaremos al punto de control:

ControlRotacion.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControlRotacion : MonoBehaviour {

  public float angulo;

  void OnTriggerEnter(Collider other) {
    MovimientoJugador jug = other.GetComponent<MovimientoJugador>();
    if (jug != null) {
      jug.Rotar(angulo);
    }
  }
}
```

Como se puede apreciar en el código, se trata de un script que declara una variable para almacenar la rotación que tendrá que efectuar el jugador, y que cuando detecta la colisión con este, llama al método *Rotar()* pasando la rotación como argumento. La variable de rotación está declarada como pública para que se pueda modificar con facilidad desde el entorno de Unity, ya que cada instancia de un punto de control tendrá almacenada una rotación distinta.

El método *Rotar()* lo tendrá accesible el jugador en su script *MovimientoJugador.cs*, el cual tendremos que modificar de la siguiente manera:

MovimientoJugador.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovimientoJugador : MonoBehaviour {

   public float vel;
```

```
   void Update () {
      transform.Translate(0, 0, vel * Time.deltaTime);
   }

   public void Rotar(float angulo) {
      StartCoroutine(RotarCorutina(angulo));
   }

   private IEnumerator RotarCorutina(float angulo) {
      vel = 0;

      if (angulo < 0) {
         for (int i = 0; i != angulo; i = i - 15) {
            transform.Rotate(new Vector3(0, -15, 0));
            yield return new WaitForSeconds(0.1f);
         }
      }

      else if (angulo > 0) {
         for (int i = 0; i != angulo; i = i + 15) {
            transform.Rotate(new Vector3(0, 15, 0));
            yield return new WaitForSeconds(0.1f);
         }
      }

      vel = 8.0f;
   }
}
```

El método *Update()* se ha simplificado respecto al juego original a una sóla línea, en la que hacemos avanzar al jugador hacia adelante. En cuanto al método *Rotar()* que llamará cada punto de control, llama a su vez a una corutina, que utilizaremos para implementar la animación de giro del jugador mediante retardos sin pausar el resto de la escena, de forma similar a como se hizo con la animación de los enemigos al desaparecer.

Esta pequeña animación tiene como objetivo que el giro no se realice inmediatamente, porque de ocurrir así podría desorientar al jugador, sobretodo teniendo en cuenta los puntos de control son invisibles para él. En su lugar, se detiene su movimiento y se hace el giro progresivamente en intervalos de 15º cada décima de segundo. De esta forma percibirá mejor que está rotando sobre si mismo.

En cuanto al script *MirarAlrededor.cs*, que teníamos asociado al jugador y también a la cámara, lo eliminaremos de ambos, ya que el movimiento de cabeza ya no será controlado con el ratón , sino por el giroscopio del dispositivo de realidad virtual, como veremos más adelante.

Ahora que dependemos por completo de los puntos de control para decidir el movimiento, los repartiremos por la escena asignando a cada uno el valor de rotación adecuado para que el recorrido pase por las zonas más importantes y vuelva al comienzo, como muestra la vista aérea de la escena en la figura 7.4.

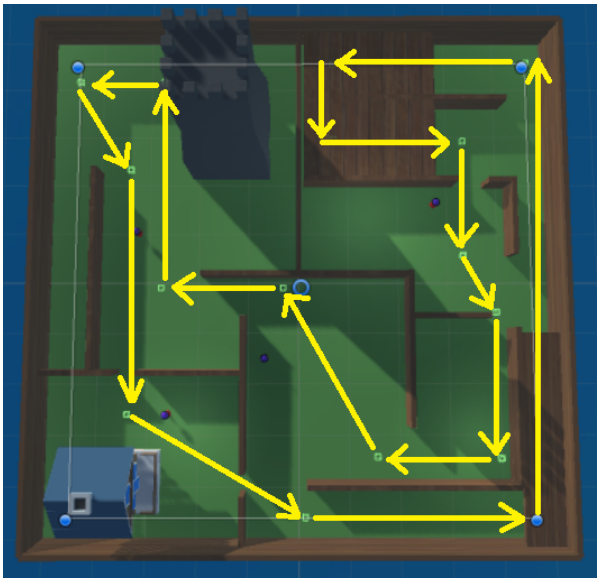


Figura 7.4: Recorrido determinado por puntos de control.

Por otra parte, también marcaremos la casilla *isTrigger* del *CapsuleCollider* de los enemigos. De esta forma podremos seguir detectando colisiones con ellos, pero no desplazarán al jugador si llegan a chocar con él. Con el recorrido establecido el jugador dará vueltas hasta que se cumpla la condición de victoria o de derrota. Sin embargo, habrá que hacer algunas modificaciones más para que pueda disparar proyectiles.

La implementación de los disparos se hará de la misma forma que en el caso de los enemigos, instanciando el prefab de un proyectil. Para ello modificaremos el método *Update()* del script *ControlJugador.cs*, y añadiremos el método *AnotarPunto()*:

ControlJugador.cs

```
[...]

void Update() {
  if (GvrViewer.Instance.Triggered) {
    Vector3 centro = new Vector3(_camara.pixelWidth / 2,
                                 _camara.pixelHeight / 2, 0);
    Ray ray = _camara.ScreenPointToRay(centro);
    RaycastHit hit;
    if (Physics.Raycast(ray, out hit)) {
      _proyectil = Instantiate(proyectilPrefab) as GameObject;
      _proyectil.transform.position =
              transform.TransformPoint(new Vector3(0, 1.5f, 0.5f));
      _proyectil.transform.LookAt(hit.point);
    }
  }

}

public void AnotarPunto() {
  _puntos++;
  _scriptControlHUD.ActualizarPuntos(_puntos, _maxPuntos);
  if (_puntos >= _maxPuntos) {
    _scriptControlHUD.FinPartida(true);
  }
}
```

En *Update()*, se sigue una lógica muy similar a la de los disparos enemigos, con algunas diferencias. En primer lugar, en cada iteración comprobamos si el gatillo del visor de Cardboard ha sido presionado, consultando el booleano *GvrViewer.Instance.Triggered*. En caso afirmativo, proyectamos un raycast hacia el centro de la cámara, en vez de simplemente hacia adelante, como hacen los enemigos. Este punto se calcula dividiendo entre 2 las propiedades pixelWidth (ancho) y pixelHeight (alto) del GameObject de la cámara, para obtener las componentes X e Y respectivamente. Si el raycast colisona con cualquier objeto, se instanciará el prefab del proyectil frente al jugador y se le asignará la orientación necesaria para que apunte hacia el punto de colisión, lo cual se especifica con *transform.LookAt(hit.point)*. De esta forma conseguimos que el proyectil siga la misma dirección que el raycast, justo hacia donde el jugador esté mirando.

El método *AnotarPunto()* se ha añadido para actualizar los puntos al derrotar enemigos. La lógica para anotar los puntos es exactamente la misma que antes, pero es necesario moverla a su propio método porque ahora es el proyectil el que detectará la colisión con los enemigos, no el jugador. Por tanto, modificaremos el script *ControlProyectil.cs* para que también detecte la colisión con enemigos, y entonces haga la llamada a *AnotarPunto()*:

ControlProyectil.cs

```
[...]
void OnTriggerEnter(Collider other) {
   ControlJugador jug = other.GetComponent<ControlJugador>();
   IAEnemigo enemigo = other.GetComponent<IAEnemigo>();

   if (jug != null) {
      jug.RecibirDisparo(fuerza);
   }

   if (enemigo != null && enemigo.vidas > 0) {
      enemigo.RecibirGolpe();
      GameObject j = GameObject.Find("Jugador");
      ControlJugador ctrlJugador = j.GetComponent<ControlJugador>();
      ctrlJugador.AnotarPunto();
   }

   Destroy(this.gameObject);
}
```

Como se puede ver el en script, se ha trasladado la lógica que se seguía anteriormente al golpear enemigos con el bastón al caso en que un proyectil impacta con un enemigo. Si el objeto alcanzado es realmente un enemigo y todavía le quedan vidas, este ejecuta el método *RecibirGolpe()*. A continuación buscamos con el método *Find()* al jugador para que se anote un punto. Por último, el objeto del proyectil se elimina de la escena con *Destroy()*.

7.2.2. Visión estereoscópica

A partir de este punto, disponemos de varias opciones para convertir la aplicación a la realidad virtual. Si hubiésemos importado los paquetes *Legacy*, una forma de hacerlo (ya obsoleta) sería sustituir la cámara principal, que tenemos asociada al jugador, por el prefab *GrvMain* (en la ruta GoogleVR/Legacy/Prefabs), que hace las funciones de cámara estereoscópica, y

ajustar otros elementos a partir de ahí. Sin embargo, debido a que los componentes en Legacy serán previsiblemente sustituidos por los más nuevos, que se mostrarán continuación, utilizaremos estos últimos para realizar la adaptación.

Usando los nuevos paquetes tenemos dos opciones, tal y como se describe en el sitio web de Google VR [45]: añadir a la escena el prefab *GvrViewerMain* (en GoogleVR/Prefabs), o bien vincular el script *GvrViewer* (en GoogleVR/Scripts) a un elemento existente (en nuestro caso lo vincularíamos a la cámara principal). En este caso se ha optado por la primera opción, aunque el resultado de ambas alternativas será el mismo.

Añadir cualquiera de estos componentes convierte la cámara principal que ya tenemos en la escena en una cámara estereoscópica, y además nos permite emular el comportamiento de un visor Cardboard o Daydream desde el editor de Unity. Si ejecutamos la aplicación desde el editor, comprobaremos que la visión ha cambiado a modo estereoscópico (figura 7.5). Podemos emular el movimiento de la cabeza si mantenemos pulsado la tecla Alt del teclado y movemos el ratón, así como la pulsación del gatillo de Cardboard (o el controlador de Daydream) si pulsamos el botón izquierdo del ratón. Recordemos que con los cambios realizados en esta versión de la aplicación para Cardboard, el jugador sólo podrá mover la cabeza y disparar, mientras que su desplazamiento estará determinado por el recorrido establecido.

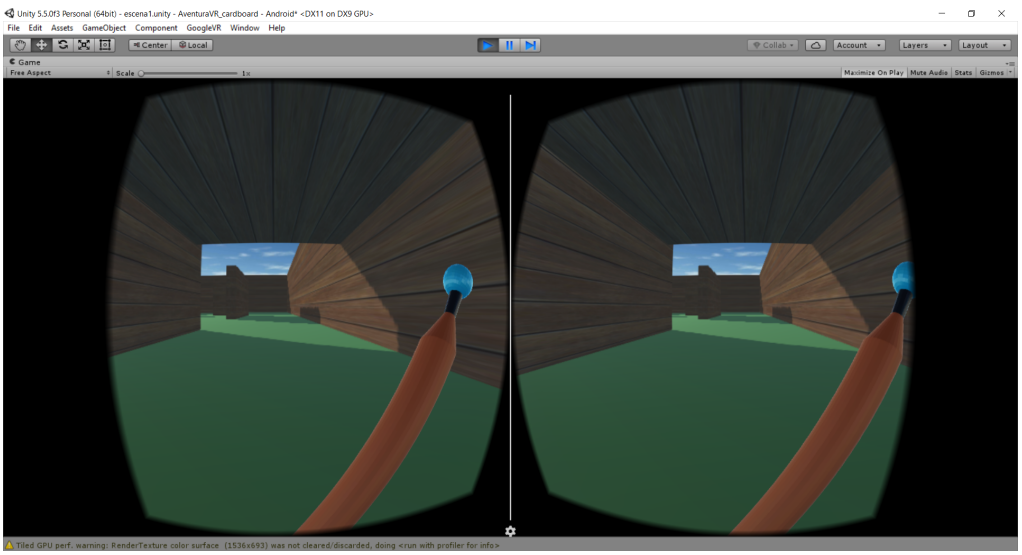


Figura 7.5: Ejecución de la aplicación en modo estereoscópico.

7.2.3. HUD y menú de opciones para visores Cardboard con gatillo

Por último, también habrá que hacer cambios en la configuración del HUD para que se muestre correctamente en modo estereoscópico. Si seleccionamos el objeto de tipo Canvas que representa el HUD, podemos observar que dejamos la propiedad *Render Mode* a su valor por defecto, *Screen Space: Overlay*. En este modo, el HUD está separado de la escena y se sitúa sobre la pantalla, cubriéndola por completo. Este comportamiento es el esperado con una cámara estándar, pero en una cámara estereoscópica cada elemento del HUD debería mostrarse independientemente para cada ojo y dentro del rango de visión, como ocurre con el resto de la escena. En modo Overlay, un icono situado en la esquina superior izquierda, por ejemplo, sólo se mostraría para el ojo izquierdo, o incluso podría renderizarse en una de las zonas oscuras donde no se vería en absoluto.

Para solucionarlo, haremos que el HUD forme parte de la escena cambiando la propiedad *Render Mode* a *World Space*. De esta forma, será visible como cualquier otro objeto del mundo, por tanto situaremos el Canvas como hijo de la cámara principal en la jerarquía, y lo moveremos para que esté justo en frente de ella, como muestra la figura 7.6. Con esta nueva distribución, es como si el jugador tuviera una pantalla holográfica frente a él que le siguiera a todas partes.

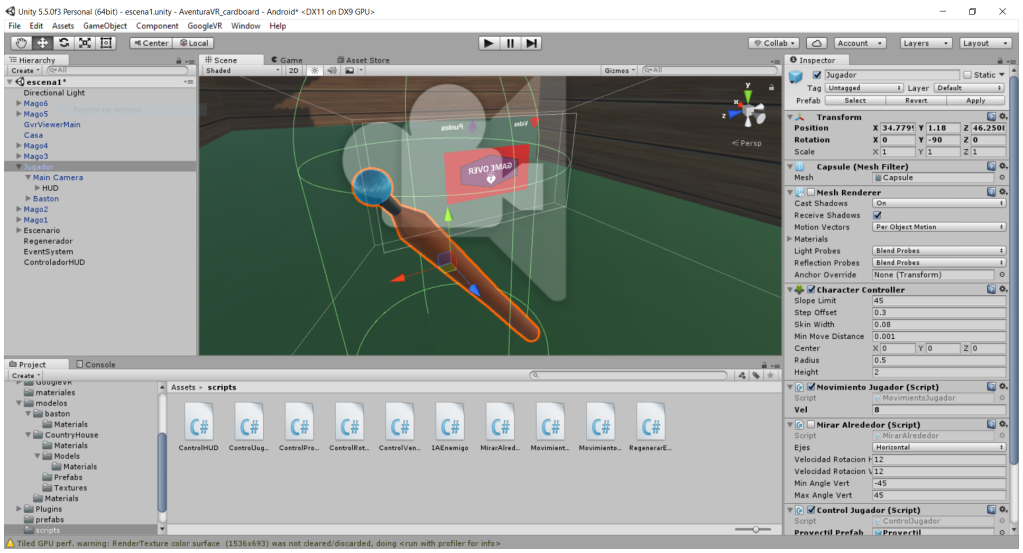


Figura 7.6: Nueva situación del HUD frente a la cámara principal.

Ahora el HUD se mostrará correctamente, y añadiremos un nuevo icono

en su centro que servirá como objetivo para apuntar. Para ello seguiremos el mismo procedimiento que para el resto de los iconos, añadiendo un objeto de tipo Image al Canvas (mediante *GameObject* → *UI* → *Image*) y asociándole un sprite que tendremos que importar previamente [19]. El resultado final se muestra en la figura 7.7. Hay que tener en cuenta que en el editor de Unity algunos iconos pueden parecer descentrados debido a la deformación de la imagen, pero en el dispositivo se mostrarán en la posición prevista.

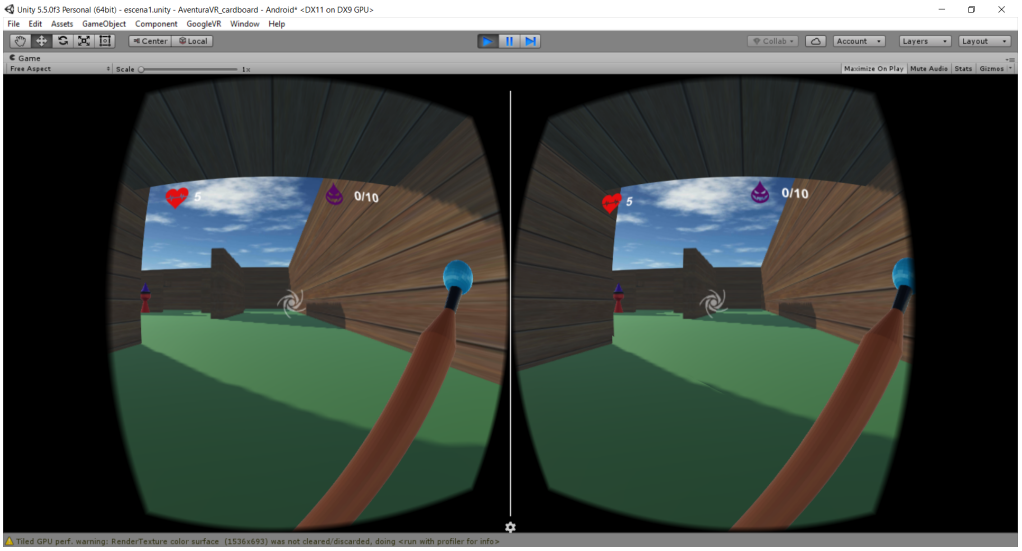


Figura 7.7: HUD integrado en cámara estereoscópica.

Por otra parte, también incorporaremos a la aplicación el menú de opciones. Dado que esta versión está diseñada para ser utilizada en visores *Cardboard* con un solo gatillo, el usuario tendrá que abrir el menú tocando la pantalla, normalmente antes de introducir el teléfono en el visor. Para ello, se montará el menú utilizando un Canvas en modo *Overlay*, es decir, de la misma forma que se hizo en la versión para escritorio. Puesto que este menú se utilizará con el dispositivo en la mano, no se mostrará en modo estereoscópico. El botón que abrirá el menú se mostrará en todo momento, en una zona de la pantalla que queda fuera del campo de visión en VR. El resultado se muestra en la figura 7.8.

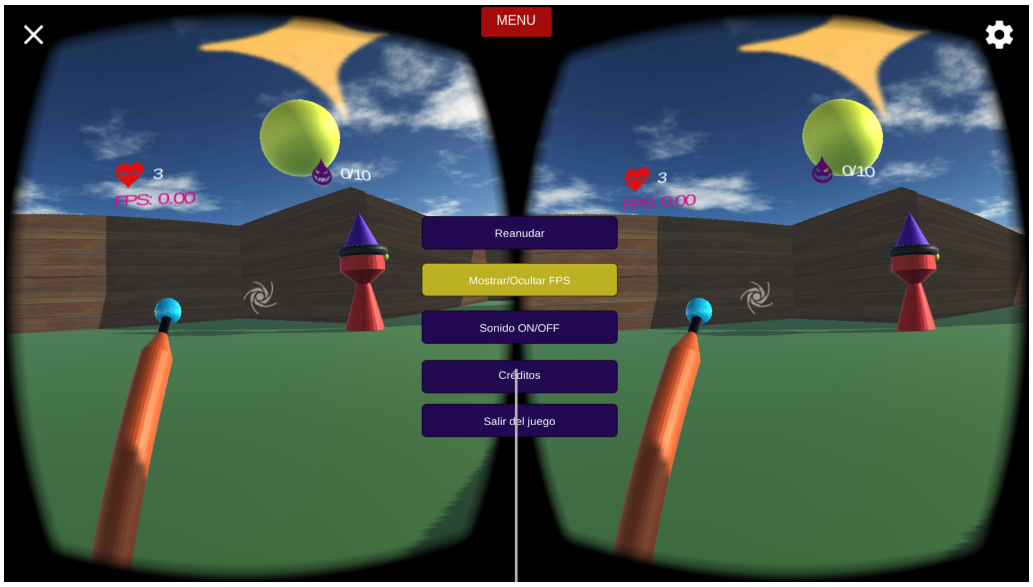


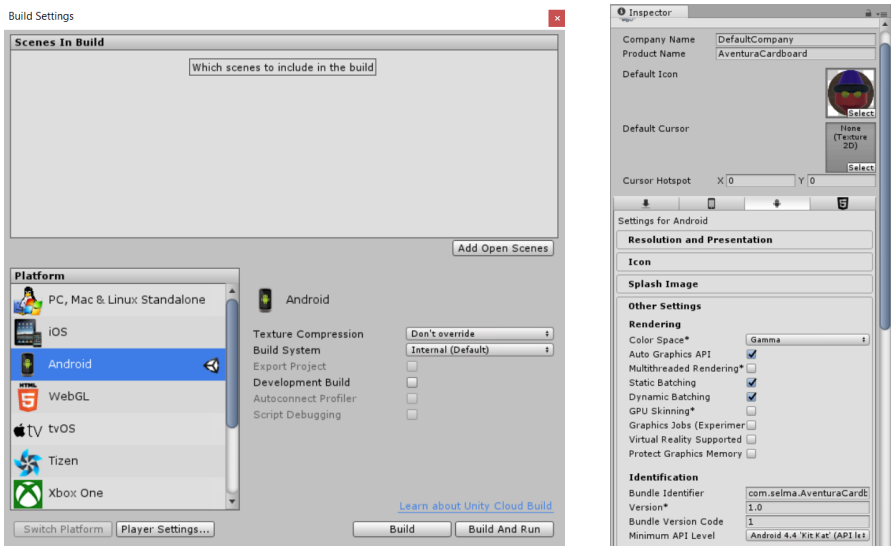
Figura 7.8: Menú de opciones en la versión para visores Cardboard con gatillo.

7.2.4. Compilación y despliegue en dispositivo

Para compilar la aplicación y poder ejecutarla en el dispositivo, que en este caso será un teléfono móvil con sistema operativo *Android*, en primer lugar tendremos que cambiar el proyecto de Unity a esta plataforma, si es que no está establecido ya.

Para ello, seleccionamos *File* → *Build Settings* y aparecerá una ventana desde donde podremos configurar todo lo relacionado con la compilación de la aplicación (figura 7.9a). Aquí seleccionaremos *Android* y después el botón *Switch Platform* para cambiar el proyecto a esta plataforma.

Al cambiar de plataforma es posible que se requiera la importación de paquetes adicionales, dependiendo de la versión de Unity y los assets del SDK que se estén utilizando. Tras finalizar el proceso, desde la misma ventana pulsamos el botón *Player Settings*. En la zona del inspector aparecerán muchas más opciones clasificadas por categorías y por plataforma. En la categoría *Other Settings* para Android (figura 7.9b), podemos encontrar la casilla *Virtual Reality Supported*, que deberíamos marcar si estuviéramos utilizando una versión de Unity con VR integrada de forma nativa. Marcar esta casilla permite a su vez elegir el dispositivo en el que se ejecutará la aplicación, como por ejemplo Oculus, Cardboard o Daydream. Sin embargo, dado que en nuestro caso hemos instalado el SDK de forma independiente, esta casilla



(a) (b)

Figura 7.9: Selección de plataforma (a) y opciones disponibles en *Player Settings* (b).

estará desmarcada por defecto y la dejaremos así. De esta forma el ejecutable final soportará igualmente la realidad virtual, pero a través del SDK importado.

En *Bundle Identifier* tendremos que introducir un nombre de paquete, con el formato *com.nombreDominio.nombreAplicacion*, y en *Minimum API Level* la versión mínima de Android para la que daremos soporte. Es importante considerar el uso de Cardboard requiere como versión mínima Android 4.4 'Kit Kat' (API level 19) y Daydream la versión 7.0 'Nougat' (API level 24). Estas son las opciones más importantes para compilar la aplicación [46], aunque hay muchas otras que conviene revisar o que puede resultar interesante modificar, como el nombre de la propia aplicación, su icono o la orientación de la pantalla.

Tras terminar con los parámetros de configuración, pulsaremos el botón *Build* para seleccionar la ruta donde se generará un fichero de tipo *apk* (Android Package Kit). Este es el fichero que tendremos que transferir al dispositivo móvil, donde lo seleccionaremos para instalar la aplicación. También es recomendable tener instalada la aplicación *Cardboard*, que puede descargarse gratuitamente desde la Play Store (figura 7.10), o desde la Apple Store en el caso de dispositivos con sistema operativo iOS.

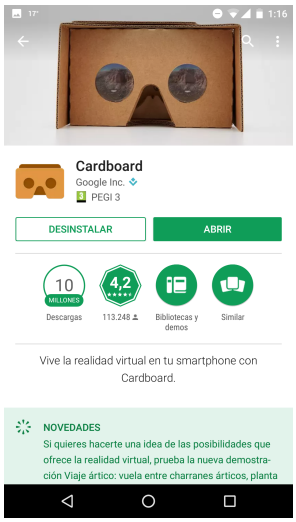


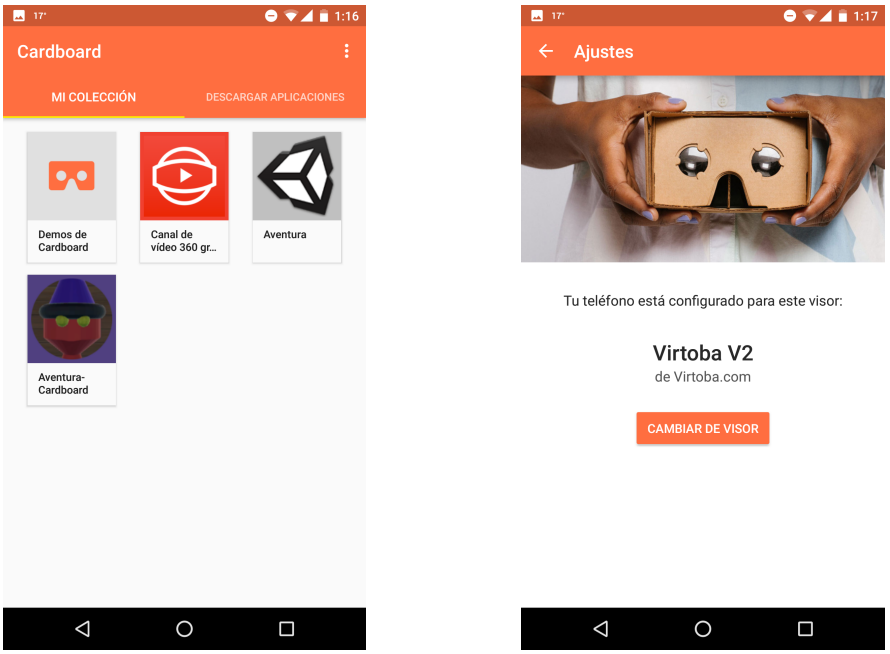
Figura 7.10: Aplicación Cardboard en Play Store.

La aplicación Cardboard consiste básicamente en un lanzador de todas las aplicaciones instaladas en el dispositivo con soporte para esta plataforma (figura 7.11a), además de una sección que permite descubrir aplicaciones adicionales. No es necesario usar el lanzador para ejecutar las aplicaciones instaladas, sin embargo es especialmente útil para poder cambiar entre distintos visores. Esto se puede hacer utilizando la cámara del dispositivo móvil para escanear el código QR que tienen impreso los visores de Cardboard. Tras enfocar el código, la aplicación Cardboard optimizará la experiencia para el visor que esté seleccionado (figura 7.11b).

Tras lanzar la aplicación, ya podemos introducir el teléfono en el visor y jugar. Para cerrarla, podemos pulsar el botón de la rueda que aparece en la esquina superior derecha para volver al lanzador. Desde ahí, podemos abrir el panel de aplicaciones abiertas con el botón cuadrado (esto puede variar según la versión del sistema operativo) y arrastrarla para cerrarla definitivamente.

7.3. Adaptación a visor con controlador

A continuación se mostrará como adaptar la aplicación a un visor que disponga de un controlador propio. En este caso el proceso de adaptación no requerirá hacer tantos cambios como en los visores Cardboard, ya que al tener disponibles un joystick y varios botones, la jugabilidad original se



(a) (b)

Figura 7.11: Lanzador de Cardboard (a) y visor seleccionado (b).

puede mantener intacta. El jugador podrá moverse de nuevo libremente utilizando el joystick, y al mismo tiempo podrá mirar hacia donde quiera de forma independiente.

Para ello se ha optado por utilizar un visor *Pasonomi VR* como el que muestra la figura 7.12. Este visor es compatible con aplicaciones Cardboard, y se puede utilizar junto con un controlador que se conecta al dispositivo móvil mediante *Bluetooth*. Visores similares a este están disponibles por unos 20 euros aproximadamente. Sus principales ventajas respecto a los visores de cartón es que son más duraderos y suelen tener reguladores para ajustar la distancia focal o la separación entre las lentes. Sin embargo, lo más interesante considerando las aplicaciones es que el uso del controlador nos da más libertad de acción al disponer de varios botones.

Partiendo de nuevo del proyecto de Unity original, en el que el jugador puede golpear a los enemigos con el bastón, importaremos los paquetes del SDK de Google VR de la misma manera que hicimos en la adaptación a los visores Cardboard.

Figura 7.12: Visor Pasonomi VR.

Una vez tengamos los paquetes importados, los únicos cambios que realizaremos en el juego original se harán para convertir el movimiento existente del jugador (basado en el teclado y el ratón) en un movimiento con controlador usando una cámara de realidad virtual.

Si recordamos la configuración del jugador y de la cámara, esta es un objeto hijo del objeto jugador en la jerarquía, por tanto la cámara sigue al jugador y realiza sus mismos movimientos. Por otra parte, tenemos un script asociado a ambos objetos, *MirarAlrededor.cs*, que controla de forma independiente su rotación con el ratón. Si movemos el ratón en el eje horizontal, el jugador (y por tanto, también la cámara) giran de izquierda a derecha. Si movemos el ratón en el eje vertical, sólo la cámara se mueve de arriba a abajo.

Si nos limitáramos a convertir la cámara existente en una cámara de realidad virtual como se hizo con el visor Cardboard, el script *MirarAlrededor.cs* no tendría ningún efecto, ya que no detectaría el ratón, mientras que el movimiento de cabeza únicamente afectaría a la cámara. El desplazamiento mediante el teclado se trasladaría al controlador, de forma que el jugador podría moverse, pero la dirección de movimiento no se vería afectada por la dirección de la mirada.

Por estos motivos, tendremos que cambiar la configuración inicial. En esta ocasión nos basaremos únicamente en el movimiento de cabeza que detecta

el giroscopio del dispositivo y en los botones del controlador. La cámara, controlada por el giroscopio del dispositivo, será la que determine la rotación del jugador, de forma que los movimientos de desplazamiento con el joystick serán relativos a la dirección en la que se esté mirando. Este es el método de control más habitual en los videojuegos en primera persona.

El primer paso será mover la cámara principal en la jerarquía de objetos, para que ya no sea hija del objeto jugador. Siendo un objeto independiente, necesitaremos un nuevo script que haga que siga al jugador en todo momento. Este script, *SeguirJugador.cs*, sustituirá a *MirarAlrededor.cs*. El código se muestra a continuación:

SeguirJugador.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SeguirJugador : MonoBehaviour {

   private GameObject _jugador;

   void Start() {
      _jugador = GameObject.Find("Jugador");
   }

   void Update () {
      transform.position =
      new Vector3(_jugador.transform.position.x,
                  _jugador.transform.position.y + 0.5f,
                  _jugador.transform.position.z);
   }
}
```

De forma similar a otros scripts vistos anteriormente, buscamos el GameObject del jugador por nombre con el método *Find()*. En el método *Update()*, actualizamos la posición de la cámara para que sea igual a la del jugador, con un pequeño desplazamiento en el eje Y para que esté a la altura de lo que sería la cabeza.

En el objeto del jugador también eliminaremos el script *MirarAlrededor.cs* y lo sustituiremos por *RotarConCamara.cs*:

RotarConCamara.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RotarConCamara : MonoBehaviour {

   private GameObject _camaraVR;

   void Start () {
      _camaraVR = GameObject.Find("Main Camera");
   }

   void Update () {
      float _rotacionH = _camaraVR.transform.localEulerAngles.y;

      transform.localEulerAngles =
      new Vector3(transform.localEulerAngles.x,
      _rotacionH,
      transform.localEulerAngles.z);
   }
```

Con este script obtenemos la rotación horizontal de la cámara principal y la aplicamos al jugador. Con esta nueva disposición de objetos y los nuevos scripts, el giroscopio del dispositivo rota la cámara principal, y esta a su vez rota horizontalmente al jugador. Los controles de desplazamiento definidos en *ControlJugador.cs* siguen siendo válidos, y ahora se aplicarán al joystick del controlador. En este mismo script habrá que hacer algunos cambios: en el método *Start()* se está obteniendo una referencia a la cámara principal mediante *GetComponentInChildren()*. Este método no obtendrá la cámara correctamente con la nueva configuración, puesto que la cámara ya no es un GameObject hijo del jugador. Por tanto la obtendremos con el método *Find()*:

ControlJugador.cs

```
[...]
void Start() {
   [...]
   _camara = GameObject.Find("Main Camera").GetComponent<Camera>();
   [...]
}
[...]
```

Alternativamente, en vez de modificar *ControlJugador.cs*, podría crearse un nuevo script con estos cambios. De todas formas, si mantenemos un proyecto de Unity para cada adaptación de la aplicación, cada proyecto puede tener su propia versión del script.

Por otra parte, sustituiremos la detección del botón izquierdo del ratón por cualquiera de los dos botones de acción del controlador. Estos dos botones son accesibles mediante los identificadores *Fire1* y *Jump* respectivamente. Por tanto, sustituiremos los métodos *GetMouseButtonDown(0)* y *GetMouseButtonUp(0)* por *GetButtonDown()* y *GetButtonUp()* para cualquiera de estos dos identificadores, en el script *ControlJugador.cs*:

ControlJugador.cs

```
void Update() {
  // boton controlador pulsado
  if (Input.GetButtonDown("Fire1") || Input.GetButtonDown("Jump")) {
  [...]
}
```

Y también en *ControlBaston.cs*, donde seguimos utilizando *Quaternion.Euler()* para que se realice la conversión al sistema de coordenadas que usa Unity en el componente *Transform*:

ControlBaston.cs

```
void Update () {

  if (Input.GetButtonDown("Fire1") || Input.GetButtonDown("Jump")) {
    // Golpe
    transform.localRotation = Quaternion.Euler(35, 90, 35);
  }

  if (Input.GetButtonUp("Fire1") || Input.GetButtonUp("Jump")) {
    // Posicion por defecto
    transform.localRotation = Quaternion.Euler(35, 90, 15);
  }
}
```

7.3.1. HUD y menú de opciones para versión en RV con controlador

Por último, tendremos que hacer los mismos cambios en el HUD que hicimos adaptando la aplicación a los visores *Cardboard*, cambiando el modo de renderización del canvas a *World Space* y haciendo que sea hijo de la cámara principal en la jerarquía. La nueva disposición de los objetos después de todos los cambios se puede apreciar en la figura 7.13.

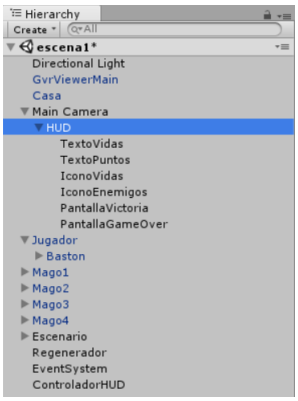


Figura 7.13: Disposición de los objetos de cámara y jugador.

En cuanto a la adaptación del menú de opciones, en esta versión disponemos de botones que nos permiten abrir el menú con un botón del controlador. Así que, a diferencia de la versión para *Cardboard*, esta vez podremos integrarlo dentro del HUD en modo estereoscópico. Para ello añadiremos los botones al Canvas existente en modo *World Space*. El menú se abrirá con su propio botón, y a partir de ese momento el usuario podrá utilizar el joystick para elegir entre las distintas opciones. Esto se muestra en la figura 7.14.

7.3.2. Compilación y despliegue

Con esto la aplicación ya está adaptada (y con la funcionalidad original) a visores con controlador, lo único que queda es compilarla y transferirla al dispositivo. De nuevo, el procedimiento es el mismo que la compilación y distribución en visores Cardboard: hay que cambiar la plataforma de trabajo a Android (o iOS, según se requiera) y configurar los parámetros en *Player Settings*, como la versión de sistema operativo, nombre e icono de la aplicación, etc.

Cuando la aplicación esté instalada en el dispositivo, tendremos que encender y vincular el controlador mediante una conexión *Bluetooth* (figura

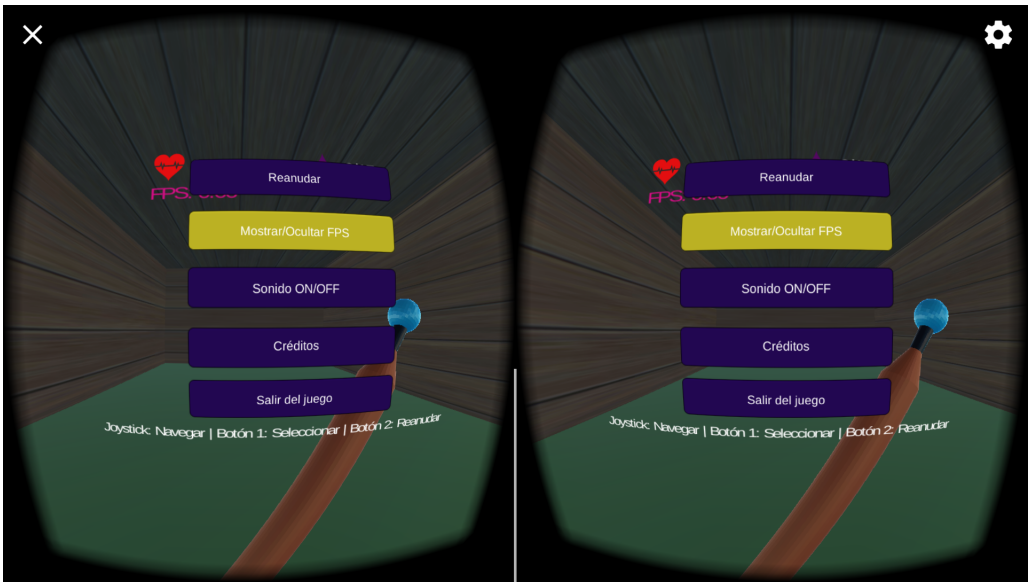


Figura 7.14: Menú de opciones en la versión en RV con controlador.

7.15).

Las figuras 7.16a y 7.16b muestran los botones y funciones disponibles en el controlador. Este tiene cuatro modos de funcionamiento asociados a los botones A, B, C, y D, aunque el único que necesitaremos será el modo C, para aplicaciones VR. Al encenderlo, el mando estará inicialmente en un modo que emula el cursor de un ratón. Para cambiar entre modos, tenemos que pulsar el botón @ seguido de la función deseada, por tanto pulsaremos @ + C y el cursor desaparecerá. En ese momento podremos lanzar la aplicación normalmente, y comprobaremos que el joystick mueve al jugador por el escenario y los botones posteriores accionan el golpe con el bastón o abren el menú (figura 7.17).

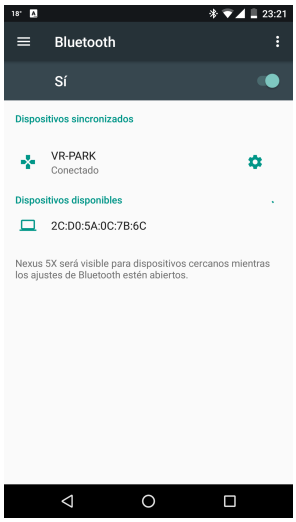
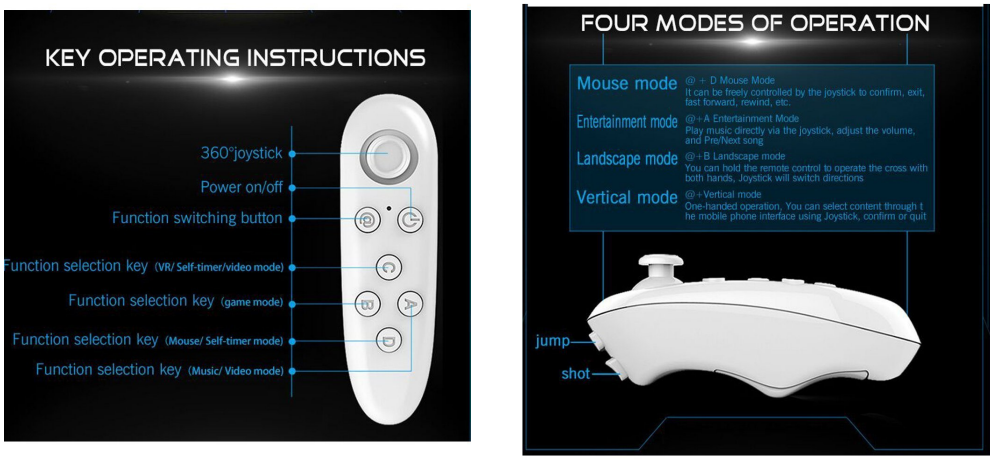


Figura 7.15: Vinculación del controlador a un teléfono Android.



(a) (b)

Figura 7.16: Botones disponibles (a) y modos de funcionamiento (b).

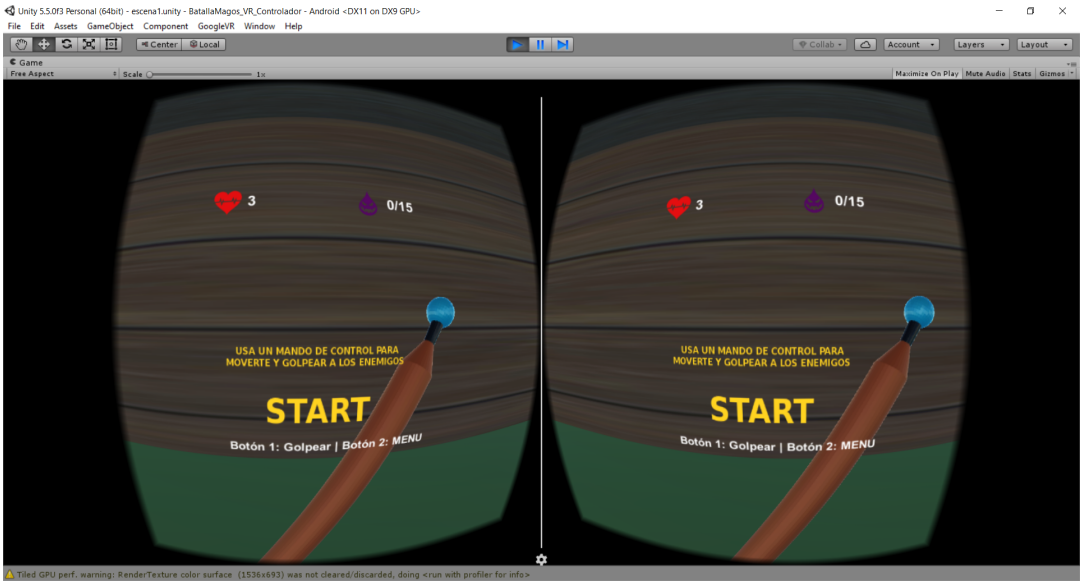


Figura 7.17: Aplicación en ejecución.

Capítulo 8

Generación de ejecutables para otras plataformas

En este capítulo se detallará el proceso de compilación y despliegue de la aplicación desarrollada en otras plataformas, algunas de las cuales tienen algunas particularidades que hay que tener en cuenta.

Como se comentó en el proceso de compilación para sistemas Android, la configuración de la compilación y despliegue de aplicaciones se encuentra en el menú *File* → *Build Settings*. Desde esté menú se puede elegir la plataforma destino en la parte izquierda y pulsar el botón *Switch Platform*, lo cual cambiará el proyecto de Unity a la plataforma especificada. En la parte derecha se pueden especificar opciones más concretas, que dependen de la plataforma, así como en el menú *Player Settings*, que es donde se encuentran la mayoría de opciones.

8.1. PC, MacOS y GNU/Linux

La versión de escritorio para PC, MacOS y GNU/Linux aparece como una misma plataforma en el selector, y en la parte derecha se puede seleccionar para cuál de las tres se quiere generar el ejecutable. Hay muchas opciones comunes en *Player Settings* para todas las plataformas, como elegir el nombre del ejecutable o seleccionar un icono, entre otras.

Para el caso concreto de Windows, también es posible seleccionar la arquitectura entre x86 y x86_64 para sistemas Windows de 64 bits. Tras pulsar el botón *Build*, Unity generará el ejecutable en el directorio seleccionado, junto con otro directorio <nombre de la aplicación>_Data, que contiene los

recursos necesarios.

El proceso para MacOS es el mismo, y además se puede generar el ejecutable correspondiente tanto si se compila con Unity en Windows como en el propio MacOS. Unity generará un directorio que permite instalar la aplicación en este sistema operativo. La aplicación se ha probado de esta manera en MacOS 10.2 (figura 8.1).

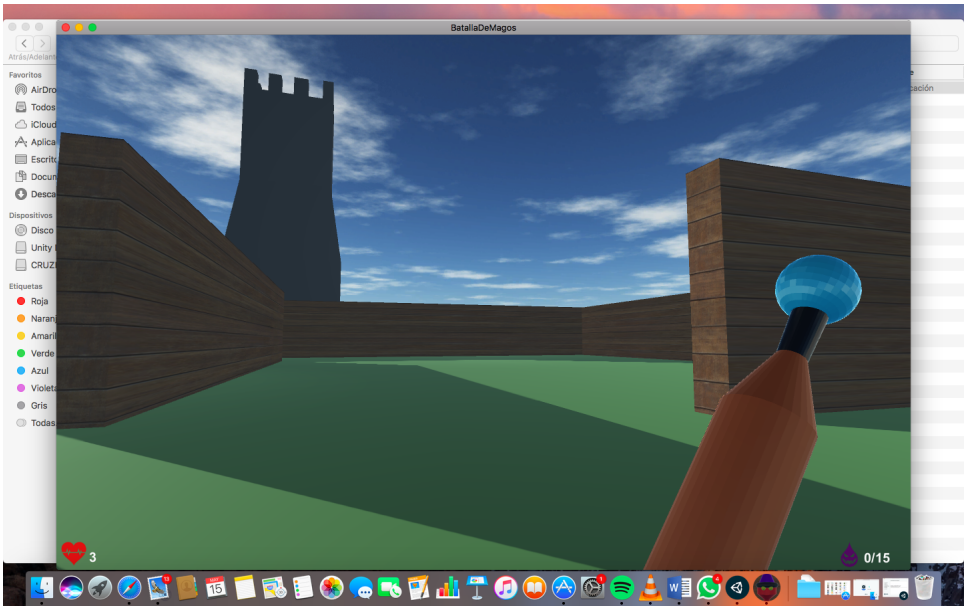


Figura 8.1: Ejecución de la aplicación en MacOS 10.2.

Para GNU/Linux también se genera el ejecutable acompañado de un directorio con los recursos necesarios. Es posible generar versiones para x86 y x86_64. Para correr la aplicación, tendremos que otorgar permisos de ejecución al fichero generado, y finalmente podremos lanzar la aplicación desde un terminal (figuras 8.2 y 8.3).

8.2. Android

El proceso de compilación para Android se ha detallado en el capítulo de adaptación de la aplicación original a la realidad virtual para Android y iOS. Sin embargo, para poder probar la aplicación en dispositivos móviles sin que sea necesario un visor de realidad virtual, se ha desarrollado una nueva versión para estas plataformas, que utiliza la pantalla táctil y el acelerómetro

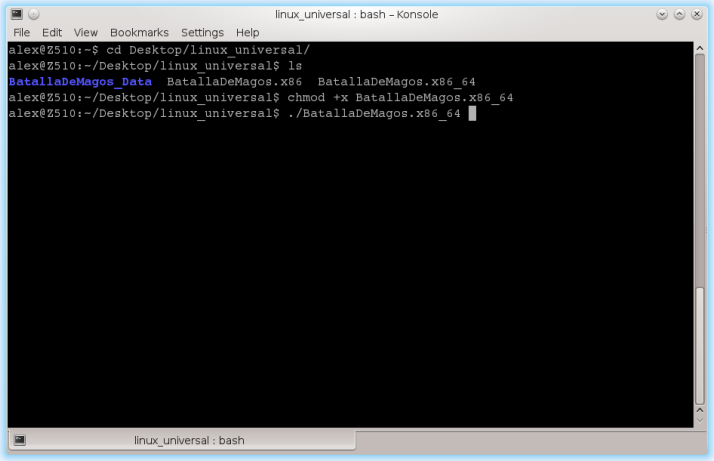


Figura 8.2: Lanzamiento de la aplicación desde terminal de Linux.

del dispositivo.

Esta versión es la misma que la versión original para escritorio, con los controles cambiados para que se pueda golpear tocando en cualquier punto de la pantalla, y desplazarse inclinando el dispositivo hacia adelante para avanzar y hacia los lados para girar. El menú de opciones también es accesible tocando la pantalla.

Los principales cambios para implementar el control por acelerómetro se encuentran los scripts *MovimientoJugador.cs* y *MirarAlrededor.cs*:

MovimientoJugador.cs

```
[...]
void Update () {
  // entrada por teclado
  float movZ = -Input.acceleration.z * vel;

  Vector3 vectorMov = new Vector3(0, 0, movZ);

  // no se puede superar vel. maxima en mov. diagonal
  vectorMov = Vector3.ClampMagnitude(vectorMov, vel);
  vectorMov.y = gravedad;

  vectorMov = vectorMov * Time.deltaTime;
```

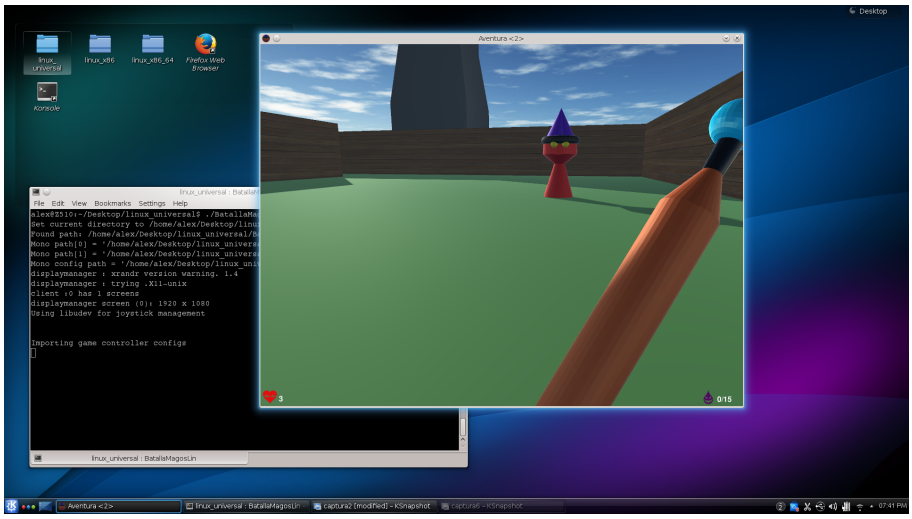


Figura 8.3: Ejecución de la aplicación en Kubuntu 14.04.

```
  // conversion a sist. global de coordenadas
  vectorMov = transform.TransformDirection(vectorMov);
  _characterController.Move(vectorMov);
}
```

MirarAlrededor.cs

```
[...]
void Update () {
  // Movimiento horizontal
  if (ejes == EjesRotacion.Horizontal) {
    transform.Rotate(0, Input.acceleration.x * velocidadRotacionH, 0);
  }
[...]
```

Como se puede ver en el código de estos scripts, el sensor del acelerómetro se lee mediante *Input.acceleration*, que se utiliza para desplazar al jugador al inclinar el dispositivo hacia adelante/atrás (eje Z) o para girar sobre si mismo al inclinarlo hacia los lados (eje X). Esta versión se muestra en la figura 8.4.

Aunque no se le de soporte oficialmente, también existe la posibilidad de emular el comportamiento de un giroscopio con el acelerómetro del dispositivo. Esto requiere acceder a Android en modo *root* e instalar un software de terceros que lo permita [48].

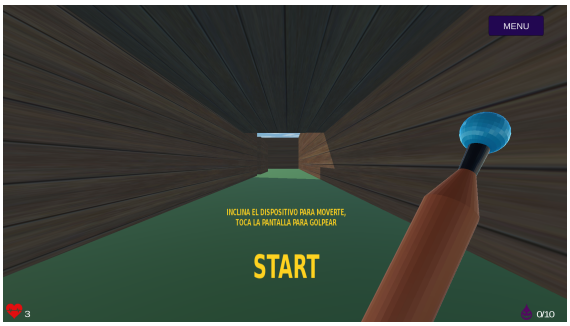


Figura 8.4: Versión de la aplicación controlada por acelerómetro.

8.3. iOS

La compilación para dispositivos iOS en Unity sigue un proceso algo distinto al del resto de plataformas. Éste se realiza en dos pasos: en primer lugar, la opción *Build* genera un proyecto *Xcode* completo con todos los ficheros necesarios. A continuación, se utiliza esta herramienta (que sólo esta disponible en MacOS) para generar el ejecutable final directamente en el dispositivo destino, que debe estar conectado en el momento de la compilación [47]. Por tanto, no es posible generar un ejecutable que pueda ser transferido posteriormente a cualquier dispositivo compatible.

Si se está utilizando la versión de Unity para Windows, se puede completar el primer paso (la generación del proyecto *Xcode*), pero la compilación y distribución del ejecutable final deberá completarse en un ordenador Mac con OS X 10.11 o posterior. Si se utiliza la versión de Unity para MacOS, después de configurar el dispositivo concreto en *Xcode* es posible realizar los dos pasos pulsando el botón *Build And Run* en *Build Settings* (figura 8.5).

8.4. WebGL

La aplicación también se ha compilado y probado en WebGL. Tras seleccionar esta plataforma y realizar la compilación, Unity genera todos los ficheros necesarios para ejecutarla como página web. Podemos lanzarla abriendo el archivo *index.html* con cualquier navegador, bien accediendo de forma local o a un servidor web en el que se hayan subido lor archivos. Se ha comprobado que ejecutarla de forma local puede provocar un fallo por cuestiones de seguridad en algunos navegadores, como Google Chrome. Una forma de solucionar esto es añadir como opción de lanzamiento *–allow-file-access-from-files*

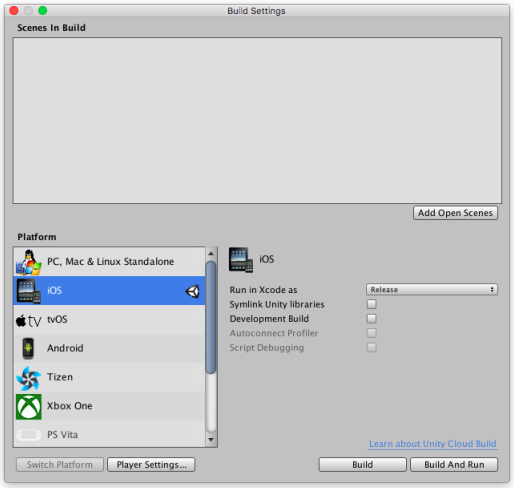
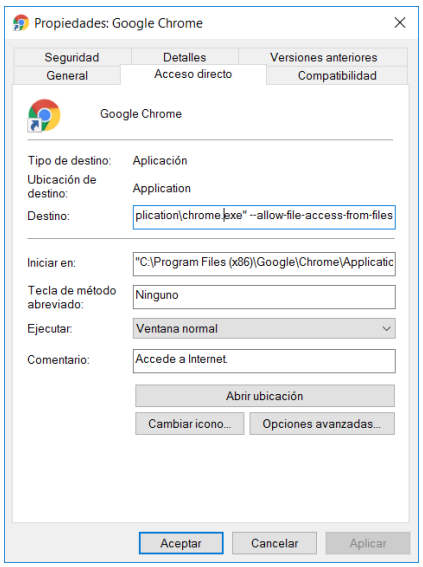


Figura 8.5: iOS seleccionado como plataforma de destino.

al acceso directo de Chrome (figura 8.6a), abrir la aplicación mediante este acceso directo y finalmente abrir (con Ctrl+O) el fichero local *index.html*. La aplicación corriendo con WebGL se muestra en la figura 8.6b.



(a)

(b)

Figura 8.6: Configuración de lanzamiento para Google Chrome (a) y aplicación ejecutándose en WebGL (b).

Capítulo 9

Conclusiones y trabajos futuros

En este proyecto se han explorado las distintas plataformas soportadas por Unity, incluyendo las opciones disponibles de realidad virtual, a través del desarrollo de varias versiones de una aplicación interactiva en 3D.

Se han implementado cuatro versiones distintas, para poder aprovechar las capacidades de distintas plataformas:

- Una versión como aplicación de escritorio, pensada para manejarse con teclado y ratón, y que tiene como plataformas objetivo los sistemas operativos Windows, MacOS, GNU/Linux y también la plataforma WebGL.
- Una versión de realidad virtual para ser utilizada con visores Cardboard con gatillo, utilizando teléfonos móviles con sistema operativo Android o iOS.
- Una versión de realidad virtual para Cardboard que se maneja con un controlador separado conectado al teléfono móvil, para Android o iOS.
- Una versión para dispositivos móviles Android o iOS (teléfonos o tablets principalmente) que se controla mediante la pantalla táctil y el acelerómetro del dispositivo.

Se ha podido comprobar que Unity ofrece muchas opciones para generar ejecutables para múltiples dispositivos y que es una herramienta adecuada para ello, no sólo por la portabilidad de proyectos entre plataformas, sino también por la flexibilidad para hacer pruebas y probar cambios rápidamente desde el editor.

Sin embargo, también se ha comprobado que hay que realizar un esfuerzo adicional a la hora de convertir una aplicación a otros dispositivos, sobretodo prestando especial atención a aspectos como el método de control en función de los métodos de entrada, licencias de desarrollo, resoluciones disponibles u otras características específicas de cada plataforma. Esto se une a otras consideraciones de diseño a la hora de desarrollar aplicaciones para la realidad virtual. Esto puede hacer que la adaptación no sea trivial.

La aplicación se ha probado en varias plataformas:

Plataformas	*Especificaciones del dispositivo*
Windows, Linux, WebGL	Lenovo Intel Core i7 4702MQ @2.2GHz, Nvidia GT740M, 8GB RAM
MacOS	Macbook Air Intel Core i5 5250U @1.6GHz, Intel HD Graphics 6000, 8GB RAM
Android	LG Nexus 5X ARMv8 @1.8GHz, Adreno 418, 2GB RAM

Por otra parte, la opción incorporada en el menú de opciones que muestra la tasa de frames por segundo (FPS) permite hacer una evaluación del rendimiento de las distintas versiones de la aplicación en varias plataformas. En los dispositivos en los que se ha ejecutado la aplicación, la tasa de refresco se ha mantenido estable en torno a los 60 FPS en todas las versiones, exceptuando las versiones en RV, donde en algunos momentos bajaba momentáneamente a 30 FPS. En las versiones de escritorio existe la ventaja de poder elegir algunos parámetros gráficos antes de lanzar la aplicación, como la resolución. Esto puede afectar al rendimiento considerablemente. En cualquier caso, la experiencia general ha sido fluida con una resolución de 1920x1080 y calidad de los gráficos máxima, lo cual se atribuye al hardware utilizado y al hecho de que los modelos 3D empleados no tienen una gran cantidad de polígonos, así como al acabado gráfico general, que se diseñó de una forma minimalista para evitar problemas de rendimiento posteriores, especialmente en dispositivos móviles.

En cuanto a las posibles ampliaciones o mejoras, la aplicación desarrollada consiste en un nivel que sirve como una demostración de una aplicación

3D, y se podría continuar añadiendo más niveles, nuevas mecánicas (como la recogida de items o la incorporación de otros obstáculos) una dificultad que fuera aumentando de forma progresiva o incluso un modo multijugador. En definitiva, el nivel puede ser una buena base para el desarrollo de un videojuego completo de mayor duración.

Existen otras posibilidades que se podrían abordar en el campo de la realidad virtual, como es incorporar a las aplicaciones la capacidad de cambiar entre modo normal y estereoscópico en tiempo de ejecución [49]. Para ello, y dependiendo de cada plataforma, podría ser necesario hacer cambios adicionales en el modo de control del jugador.

En el caso de la versión para visores Cardboard con gatillo, se podrían explorar implementaciones alternativas a la utilizada mediante puntos de control, como hacer que el jugador se desplazara hacia donde esté mirando, siguiendo en todo momento la dirección de la cámara. La gestión del menú también podría cambiarse, para que apareciera al mantener pulsado un botón durante unos segundos, o para poder seleccionar las opciones con la propia mirada.

Por último, respecto al propio proyecto, se considera que se han cubierto los principales objetivos propuestos, mostrando las capacidades de Unity como herramienta de desarrollo, y explorando su capacidad multiplataforma mediante la implementación de distintas versiones adaptadas a varios dispositivos, con especial énfasis en aquellos que hacen uso de realidad virtual.

Bibliografía

[1] Disponible en <https://unity3d.com/es/public-relations>
[Fecha de consulta: 12/06/2017]

[2] Sitio web de Unreal Engine
Disponible en
<https://www.unrealengine.com/what-is-unreal-engine-4>
[Fecha de consulta: 04/05/2017]

[3] Sitio web de CryEngine
Disponible en <https://www.cryengine.com/>
[Fecha de consulta: 04/05/2017]

[4] Disponible en <https://unity3d.com/es/unity/multiplatform/>
[Fecha de consulta: 02/05/2017]

[5] Disponible en
<https://en.wikipedia.org/wiki/Unity_(game_engine)>
[Fecha de consulta: 27/12/2016]

[6] Tutoriales y documentación de Unity
Disponible en <https://unity3d.com/es/learn>
[Fecha de consulta: 04/05/2017]

[7] Uso de API Vulkan en Unity
Disponible en
<https://blogs.unity3d.com/es/2016/09/29/introducing-the-vulkan-renderer-preview/>
[Fecha de consulta: 05/05/2017]

[8] Getting started with WebGL development

Disponible en
<https://docs.unity3d.com/Manual/webgl-gettingstarted.html>

[9] Getting started with iOS development

Disponible en
<https://docs.unity3d.com/Manual/iphone-GettingStarted.html>

[10] Disponible en
<https://docs.unity3d.com/es/current/Manual/android-GettingStarted.html>

[11] Licencia de Unity incluída con kit de desarrollo de WiiU

Disponible en <https://venturebeat.com/2012/11/02/game-developers-start-your-unity-3d-engines-interview/>

Herramientas de desarrollo para consolas Nintendo

<https://developer.nintendo.com/tools>

[Fecha de consulta: 14/05/2017]

[12] Disponible en <https://docs.unity3d.com/Manual/tvOS.html>

[13] Hocking, Joseph. (2015). Unity in Action, Manning Publications Co., Shelter Island, NY

[14] Disponible en <http://viz.aset.psu.edu/gho/sem_notes/3d_-fundamentals/html/3d_coordinates.html)>

[Fecha de consulta: 05/01/2017]

[15] Disponible en
<http://www.gamasutra.com/blogs/SaraCasen/20160713/276970/White_-Boxing_Your_Game.php>

[Fecha de consulta: 10/01/2017]

[16] Disponible en
<https://unity3d.com/es/learn/tutorials/topics/interface-essentials/hierarchy-and-parent-child-relationships>

[Fecha de consulta: 04/01/2017]

[17] Disponible en
<https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

[Fecha de consulta: 04/02/2017]

[18] Disponible en
<https://docs.unity3d.com/ScriptReference/Physics.Raycast.html>
[Fecha de consulta: 05/03/2017]

[19] Iconos obtenidos de <http://game-icons.net/>
[Fecha de consulta: 22/03/2017]

[20] Imágenes obtenidas de <http://www.plaintextures.com>
[Fecha de consulta: 22/03/2017]

[21] Imágenes obtenidas de <https://93i.de/p/free-skybox-texture-set/>
Autor: Heiko Irrgang
Licencia Creative Commons
[Fecha de consulta: 09/04/2017]

[22] <https://www.assetstore.unity3d.com/en/>
[Fecha de consulta: 09/04/2017]

[23] Paquete de recursos obtenido de <http://u3d.as/wgf>
Pagina web del autor <https://sketchfab.com/Bek>
[Fecha de consulta: 11/04/2017]

[24] Sonido obtenido de <https://www.freesound.org>
Autor: 'vr'
Licencia Creative Commons
<https://creativecommons.org/licenses/by/3.0/>
[Fecha de consulta: 11/04/2017]

[25] Sonido obtenido de <https://www.freesound.org>
Autor: 'Tuudurt'
Licencia Creative Commons de dominio público
<https://creativecommons.org/publicdomain/zero/1.0/>
[Fecha de consulta: 11/04/2017]

[26] Sonido obtenido de <https://www.freesound.org>
Autor: 'Kastenfrosch'
Licencia Creative Commons de dominio público
<https://creativecommons.org/publicdomain/zero/1.0/>
[Fecha de consulta: 11/04/2017]

[27] Sonido obtenido de <https://www.freesound.org>
Autor: 'Triper'
Licencia Creative Commons de dominio público
<https://creativecommons.org/publicdomain/zero/1.0/>
[Fecha de consulta: 11/04/2017]

[28] Sonido obtenido de <https://www.freesound.org>
Autor: 'vollkornbrot'
Licencia Creative Commons
<https://creativecommons.org/licenses/by-nc/3.0/>

[29] Historia de la realidad virtual
Disponible en <https://en.wikipedia.org/wiki/Virtual_reality>
[Fecha de consulta: 06/05/2017]

[30] The Cave: audio visual experience automatic virtual environment
Disponible en <http://dl.acm.org/citation.cfm?doid=129888.129892>
[Fecha de consulta: 12/06/2017]

[31] Sega VR
Disponible en <http://segaretro.org/Sega_VR>
[Fecha de consulta: 12/06/2017]

[32] Máquinas Virtuality
Disponible en <https://en.wikipedia.org/wiki/Virtuality_(gaming)>
[Fecha de consulta: 12/06/2017]

[33] <http://www.androidauthority.com/daydream-vr-ready-phones-specs-727780/>
[Fecha de consulta: 16/04/2017]

[34] Sitio web de Oculus Rift disponible en <https://www.oculus.com/>
[Fecha de consulta: 06/05/2017]

[35] Disponible en <https://www.vive.com/us/ready/>
[Fecha de consulta: 07/05/2017]

[36] Disponible en <https://en.wikipedia.org/wiki/HTC_Vive>
[Fecha de consulta: 07/05/2017]

[37] Disponible en <https://en.wikipedia.org/wiki/PlayStation_VR>
[Fecha de consulta: 07/05/2017]

[38] Especificaciones técnicas de Samsung Gear VR
Disponible en
<http://www.samsung.com/global/galaxy/gear-vr/specs/>
[Fecha de consulta: 09/05/2017]

[39] Especificaciones técnicas de Microsoft Hololens
Disponible en <https://developer.microsoft.com/en-us/windows/mixed-reality/hololens_hardware_details>
[Fecha de consulta: 09/05/2017]

[40] Acerca del soporte nativo para realidad virtual
Disponible en
<https://docs.unity3d.com/Manual/VROverview.html>

[41] SDK Google VR para Unity
<https://developers.google.com/vr/>
[Fecha de consulta: 16/04/2017]

[42] Disponible en
<https://developer.android.com/studio/index.html?hl=es-419>
[Fecha de consulta: 17/04/2017]

[43] Disponible en
<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

[44] Disponible en <https://developers.google.com/vr/unity/download>
[Fecha de consulta: 17/04/2017]

[45] Disponible en <https://developers.google.com/vr/unity/guide>
[Fecha de consulta: 17/04/2017]

[46] Disponible en <https://developers.google.com/vr/unity/get-started>
[Fecha de consulta: 18/04/2017]

[47] Compilación para iOS desde Windows utilizando XCode

Disponible en <https://unity3d.com/es/learn/tutorials/topics/mobile-touch/building-your-unity-game-ios-device-testing>

[Fecha de consulta: 16/05/2017]

[48] Uso de Google Cardboard sin giroscopio

Disponible en <http://buckydroid.com/vr-without-gyroscope/>

[Fecha de consulta: 22/05/2017]

[49] Activación de modo VR mediante VR.Settings

Disponible en <https://docs.unity3d.com/ScriptReference/VR.VRSettings-enabled.html>

[Fecha de consulta: 15/05/2017]